HAL
open science

Validation sémantique dans les théories structurées : application à un langage de programmation générique

Pascal Drabrik

To cite this version:

Pascal Drabrik. Validation sémantique dans les théories structurées : application à un langage de programmation générique. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1989. Français. NNT : . tel-00333592

HAL Id: tel-00333592

https://theses.hal.science/tel-00333592v1

Submitted on 23 Oct 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Tu8505

THESE

présentée par

DRABIK Pascal

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(*arrêté ministériel du 5 Juillet 1984*)

Spécialité : INFORMATIQUE

==========

Validation Sémantique dans les Théories Structurées : Application à un Langage de Programmation Générique

==========

Date de soutenance : 22 Novembre 1989

Composition du jury :	J. MOSSIERE	Président
	Ph. JORRAND E. PAUL	Rapporteurs
	D. BERT M. SINTZOFF	Examinateurs

Thèse préparée au sein du LIFIA
Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle

INSTITUT NATIONAL POLYTECHNIQUE
GRENOBLE

46 avenue F. Viallet - 38031 GRENOBLE Cedex -

Tél : 76.57.45.00

ANNEE UNIVERSITAIRE 1989

Président de l'Institut
Monsieur Georges LESPINARD

PROFESSEURS DES UNIVERSITES

ENSERG	BARIBAUD	Michel
ENSIEG	BARRAUD	Alain
ENSPG	BAUDELET	Bernard
INPG	BEAUFILS	Jean-Pierre
ENSERG	BLIMAN	Samuel
ENSHMG	BOIS	Philippe
ENSEEG	BONNETAIN	Lucien
ENSPG	BONNET	Guy
ENSIEG	BRISSONNEAU	Pierre
IUFA	BRUNET	Yves
ENSHMG	CAILLERI E	Denis
ENSPG	CAVAIGNAC	Jean-François
ENSPG	CHARTIER	Germain
ENSERG	CHENEVIER	Pierre
UFR PGP	CHERADAME	Hervé
ENSIEG	CHERUY	Arlette
ENSERG	CHOVET	Alain
ENSERG	COHEN	Joseph
ENSEEG	COLINET	Catherine
ENSIEG	CORNUT	Bruno
ENSIEG	COULOMB	Jean-Louis
ENSERG	COUMES	André
ENSIMAG	CROWLEY	James
ENSHMG	DARVE	Félix
ENSIMAG	DELLA DORA	Jean-François
ENSERG	DEPEY	Maurice
ENSPG	DEPORTES	Jacques
ENSEEG	DEROO	Daniel
ENSEEG	DESRE	Pierre
ENSERG	DOLMAZON	Jean-Marc
ENSEEG	DURAND	Francis
ENSPG	DURAND	Jean-Louis
ENSHMG	FAUTRELLE	Yves
ENSIEG	FOGGIA	Albert
ENSIMAG	FONLUPT	Jean
ENSIEG	FOULARD	Claude
UFR PGP	GANDINI	Alessandro
ENSPG	GAUBERT	Claude
ENSERG	GENTIL	Pierre
ENSIEG	GENTIL	Sylviane
IUFA	GREVEN	Hélène
ENSIEG	GUEGUEN	Claude
ENSERG	GUERIN	Bernard
ENSEEG	GUYOT	Pierre

ENSPG	JOST	Rémy
ENSPG	JOUBERT	Jean-Claude
ENSIEG	JOURDAIN	Geneviève
ENSIEG	LACOUME	Jean-Louis
ENSIEG	LADET	Pierre
ENSHMG	LESIEUR	Marcel
ENSHMG	LESPINARD	Georges
ENSPG	LONGEQUEUE	Jean-Pierre
ENSHMG	LORET	Benjamin
ENSEEG	LOUCHET	François
ENSEEG	LUCAZEAU	Guy
ENSIEG	MASSE	Philippe
ENSIEG	MASSELOT	Christian
ENSIMAG	MAZARE	Guy
ENSIMAG	MOHR	Roger
ENSHMG	MOREAU	René
ENSIEG	MORET	Roger
ENSIMAG	MOSSIERE	Jacques
ENSHMG	OBLED	Charles
ENSEEG	OZIL	Patrick
ENSEEG	PAULEAU	Yves
ENSIEG	PERRET	Robert
ENSHMG	PIAU	Jean-Miche
ENSERG	PIC	Etienne
ENSIMAG	PLATEAU	Brigitte
ENSERG	POUPOT	Christian
ENSEEG	RAMEAU	Jean-Jacqu
ENSPG	REINISCH	Raymond
UFR PGP	RENAUD	Maurice
UFR PGP	ROBERT	André
ENSIMAG	ROBERT	François
ENSIEG	SABONNADIERE	Jean-Clau
ENSIMAG	SAUCIER	Gabriele
ENSPG	SCHLENKER	Claire
ENSPG	SCHLENKER	Michel
ENSERG	SERMET	Pierre
UFR PGP	SILVY	Jacques
ENSHMG	SIRIEYS	Pierre
ENSEEG	SOHM	Jean-Clau
ENSIMAG	SOLER	Jean-Loui
ENSEEG	SOUQUET	Jean-louis
ENSHMG	TROMPETTE	Philippe
ENSPG	VINCENT	Henri
ENSERG	ZADWORNY	François

SITUATION PARTICULIERE

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSIMAG	LATOMBE	J.Claude	Détachement.....	21/10/1989
ENSHMG	PIERRARD	J.Marie	Détachement.....	30/04/1989
ENSIMAG	VEILLON	Gérard	Détachement.....	30/09/1990
ENSIMAG	VERJUS	J.Pierre	Détachement.....	30/09/1989
ENSPG	BLOCH	Daniel	Récteur à c/.....	21/12/1988

SURNOMBRE

INPG	CHIAVERINA	Jean	30/09/1989
ENSHMG	BOUVARD	Maurice...	30/09/1991
ENSEEG	PARIAUD	J.Charles	30/09/1991

PERSONNES AYANT OBTENU LE DIPLOME
d'habilitation à diriger des recherches

BECKER	M.	DANES	F.	GHIBAUDO	G.	MULLER	J.
BINDER	Z.	DEROO	D.	HAMAR	S.	NGUYEN TRONG	B.
CHASSERY	J.M.	DIARD	J.P.	HAMAR	R.	NIEZ	J.J.
CHOLLET	J.P.	DION	J.M.	LACHENAL	D.	PASTUREL	A.
COEY	J.	DUGARD	L.	LADET	P.	PLA	F.
COLINET	C.	DURAND	M.	LATOMBE	C.	ROGNON	J.P.
COMMAULT	C.	DURAND	R.	LE HUY	H.	ROUGER	J.
CORNUEJOLS	G.	GALERIE	A.	LE GORREC	B.	TCHUENTE	M.
COULOMB	J.L.	GAUTHIER	J.P.	MADAR	R.	VINCENT	H.
COURNIL	M.	GENTIL	S.	MEUNIER	G.	YAVARI	A.R
DALARD	F.						

CHERCHEURS DU C.N.R.S.

DIRECTEURS DE RECHERCHE CLASSE 0

LANDAU	Ioan
NAYROLLES	Bernard

DIRECTEURS DE RECHERCHE 1ère CLASSE

ANSARA	Ibrahim	KRAKOWIAK	Sacha
CARRE	René	LEPROVOST	Christian
FRUCHARD	Robert	VACHAUD	Georges
HOPFINGER	Emile	VERJUS	Jean-Pierre
JORRAND	Philippe		

DIRECTEURS DE RECHERCHE 2ème CLASSE

ALEMANY	Antoine	JOUD	Jean-Charles
ALLIBERT	Colette	KAMARINOS	Georges
ALLIBERT	Michel	KLEITZ	Michel
ARMAND	Michel	KOFMAN	Walter
AUDIER	Marc	LEJEUNE	Gérard
BERNARD	Claude	MADAR	Roland
BINDER	Gilbert	MERMET	Jean
BONNET	Roland	MICHEL	Jean-Marie
BORNARD	Guy	MEUNIER	Jacques
CAILLER	Marcel	PEUZIN	Jean-Claude
CALMET	Jacques	PIAU	Monique
CHATILLON	Christian	RENOUARD	Dominique
CLERMONT	Jean-Robert	SENATEUR	Jean-Pierre
COURTOIS	Bernard	SIFAKIS	Joseph
DAVID	René	SIMON	Jean-Paul
DION	Jean-Michel	SUERY	Michel
DRIOLE	Jean	TEODOSIU	Christian
DURAND	Robert	VAUCLIN	Michel
ESCUDIER	Pierre	VENNEREAU	Pierre
EUSTATHOPOULOS	Nicolas		

PERSONNALITES AGREEES A TITRE PERMANENT A DIRIGER DES TRAVAUX DE RECHERCHE

(DECISION DU CONSEIL SCIENTIFIQUE)

ENSEEG	HAMMOU	Abdelkader
	MARTIN-GARIN	Régina
	SARRAZIN	Pierre
	SIMON	Jean-Paul
ENSERG	BOREL	Joseph
ENSIEG	DESCHIZEAUX	Pierre
	GLANGEAUD	François
	PERARD	Jacques
	REINISCH	Raymond
ENSHMG	ROWE	Alain
ENSIMAG	COURTIN	Jacques
C.E.N.G	CADET	Jean
	COEURE	Philippe
	DELHAYE	Jean-Marc
	DUPUY	Michel
	JOUVE	Hubert
	NICOLAU	Yvan
	NIFENECKER	Hervé
	PERROUD	Paul
	PEUZIN	Jean-Claude
	TAIEB	Maurice
	VINCENDON	Marc

Laboratoire extérieurs :

C.N.E.T.	DEVINE	Rodericq
	GERBER	Roland
	MERCKEL	Gérard
	PAULEAU	Yves

XXXXXXXXXXXXXXXXX

UNIVERSITE JOSEPH FOURIER
SCIENCES. TECHNOLOGIE. MEDECINE
GRENOBLE 1

Président de l'Université :

M. NEMOZ Alain

ANNEE UNIVERSITAIRE 1988 - 1989

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ERE CLASSE

ADIBA Michel	Informatique
ANTOINE Pierre	Géologie I.R.I.G.M.
ARNAUD Paul	Chimie Organique
ARVIEU Robert	Physique Nucléaire I.S.N.
AURIAULT Jean Louis	Mécanique
AYANT Yves	Physique Approfondie
BARJON Robert	Physique Nucléaire I.S.N.
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean René	Statistiques - Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean Paul	Mathématiques Pures
BILLET Jean	Géographie
BOEHLER Jean Paul	Mécanique
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CASTAING Bernard	Physique
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FINKE Gerde	Informatique
GAGNAIRE Didier	Chimie Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GERMAIN Jean Pierre	Mécanique
GIDON Maurice	Géologie
GUITTON Jacques	Chimie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean René	Mathématiques Pures

KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean Marie	Sciences Nucléaires I.S.N.
LONGEQUEUE Nicole	Physique
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PANNETIER Jean	Chimie
PAYAN Jean Jacques	Mathématiques Pures
PEBAY PEYROULA Jean Claude	Physique
PERRIER Guy	Géophysique
PIERRE Jean Louis	Chimie Organique
RENARD Michel	Thermodynamique
RIEDTMANN Christine	Mathématiques
RINAUDO Marguerite	Chimie C.E.R.M.A.V.
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS DE 2EME CLASSE

ARMAND Gilbert	Géographie
ATTANE Pierre	Mécanique
BARET Paul	Chimie
BERTIN José	Mathématiques
BLANCHI Jean Pierre	S.T.A.P.S.
BLOCK Marc	Biologie
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BORRIONE Dominique	Automatique informatique
BOUVET Jean	Biologie
BROSSARD Jean	Mathématiques
BRUANDET Jean François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
CHOLLET Jean Pierre	Mécanique
COLOMBEAU Jean François	Mathématiques (ENSL)
COURT Jean	Chimie
CUNIN Pierre Yves	Informatique

DAVID Jean	Géographie
DHOUAILLY Danielle	Biologie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHIAN Hervé	Géographie
HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
HERINO Roland	Physique
JARDON Pierre	Chimie
KERCKHOVE Claude	Géologie
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MOREL Alain	Géographie
NEMOZ Alain	Thermodynamique C.N.R.S. C.R.T.B.T.
NGUYEN HUY Xuong	Informatique
OUDET Bruno	Mathématiques Appliquées
PAUTOU Guy	Biologie
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PELLETIER Guy	Astrophysique
PERRIN Claude	Sciences Nucléaires I.S.N.
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
REGNARD Jean René	Physique
RICHARD Jean Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Danielle	Chimie
ROBERT Gilles	Mathématiques Pures
ROBERT Jean Bernard	Chimie Physique
SARROT REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VINCENT Gilbert	Physique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L'I.U.T. 1

PROFESSEURS DE 1ERE CLASSE

BUISSON Roger	Physique I.U.T. 1
CHEHIKIAN Alain	E.E.A. I.U.T.1
DODU Jacques	Mécanique Appliquée I.U.T. 1
NEGRE Robert	Génie Civil I.U.T. 1
NOUGARET Marcel	Automatique I.U.T. 1
PERARD Jacques	E.E.A. I.U.T. 1

PROFESSEURS DE 2EME CLASSE

BEE Marc	Physique I.U.T. 1
BOUTHINON Michel	E.E.A. I.U.T. 1
CHAMBON René	Génie Mécanique I.U.T. 1.
CHENAVAS Jean	Physique I.U.T. 1
CHILO Jean	Physique I.U.T. 1
CHOUTEAU Gérard	Physique I.U.T. 1
CONTE René	Physique I.U.T. 1
FOSTER Panayotis	Chimie I.U.T. 1
GOSSE Jean Pierre	E.E.A. I.U.T. 1
GROS Yves	Physique I.U.T. 1
HAMAR Roger	Chimie I.U.T. 1
KUHN Gérard, détaché	Physique I.U.T. 1
LEVIEL Jean Louis	Physique I.U.T. 1
MAZUER Jean	Physique I.U.T. 1
MICHOULIER Jean	Physique I.U.T. 1
MONLLOR Christian	E.E.A. I.U.T. 1
PERRAUD Robert	Chimie I.U.T. 1
PIERRE Gérard	Chimie I.U.T. 1
TERRIEZ Jean Michel	Génie Mécanique I.U.T. 1
TOUZAIN Philippe	Chimie I.U.T. 1
TURGEMAN Sylvain	Génie civil
VINCENDON Marc	Chimie I.U.T. 1
ZIGONE Michel	Physique I.U.T. 1

Remerciements

Le travail que représente une thèse nécessite un effort très important. Cet effort peut néanmoins être modulé en fonction des conditions dans lesquelles il est accompli. Ainsi, l'environnement, tant professionnel que privé, dans lequel cette thèse a été préparée a eu des conséquences importantes sur le résultat.

Je tiens de ce fait à remercier

M. Philippe Jorrand, d'une part pour m'avoir accepté dans le laboratoire qu'il dirige et d'autre part pour avoir bien voulu juger et commenter ce travail, en se montrant intéressé par ce document.

M. Etienne Paul, pour avoir eu la patience d'attendre un temps non négligeable afin d'évaluer et de commenter mon travail.

M. Jacques Mossière, pour avoir bien voulu présider la soutenance de cette thèse.

M. Didier Bert, pour m'avoir accepté au sein de son équipe et pour m'avoir dirigé tout au long de ce travail de recherche.

M. Michel Sintzoff, pour m'avoir fait l'honneur de faire partie de mon jury.

Je tiens également à remercier

Melle Sylvie Rogé, pour m'avoir encouragé durant certaines périodes délicates et inévitables.

M. Rachid Echahed, avec qui j'ai eu fréquemment des discussions très intéressantes et qui a su également me pousser chaque fois que le courage me faisait défaut.

Mes collègues et amis, Marie-Laure Potet, Philippe Chatelin et James L. Crowley, Mary Cisneros, Maria-Blanca Ibáñez, Guilherme Bittencourt, Hubert Comon, Thierry Fraichard, Jean-Michel Hufflen, Denis Lugiez, Philippe Schnoebelen et Sadik Sebbar.

Je tiens à remercier finalement mes parents, Janine et Stanislas Drabik, qui se sont sans cesse sacrifiés pour que je puisse aboutir à ce résultat.

Drabik P.

Table des matières

1- Introduction 1

2- Définitions 5

2-1- Introduction 5
2-2- Signatures et présentations 5
2-3- Théories et morphismes de théories 7
2-4- Réécriture et formes normales 9
2-5- Utilisation de ces concepts 10
2-6- Techniques de démonstration 11

3- OASIS 13

3.1- Introduction 13
3.2- L'atelier OASIS 13
3.3- Liste des commandes possibles 18
3.3.1- Commandes générales de l'atelier 18
3.3.1.1- Généralités 18
3.3.1.2- Compilation et manipulation de fichiers 18
3.3.1.3- Visualisation des objets 18
3.3.1.4- Modifications des objets 18
3.3.1.5- Contrôles sur les objets 19
3.3.1.6- Validation sémantique des objets 19
3.3.1.7- Documentation en ligne 19
3.3.1.8- Divers 20
3.3.2- Fonctions du démonstrateur de théorèmes 20
3.3.2.1- Généralités 20
3.3.2.2- Construction de l'arborescence 21
3.3.2.3- Suppression dans l'arborescence 21
3.3.2.4- Visualisation et déplacement dans l'arborescence 21
3.3.2.5- Divers 21
3.4- Environnement 22
3.5- Langage 25
3.5.1- Syntaxe employée 25
3.5.2- Description d'un type 26
3.5.2.1- Généralités 26
3.5.2.2- Exemple de la pile 27
3.5.2.3- Rubriques concernant la spécification 28
3.5.2.3.1- parametres 28
3.5.2.3.2- import 28
3.5.2.3.3- constantes 29
3.5.2.3.4- operations 29
3.5.2.3.5- var_eqs 29
3.5.2.3.6- equations 29

3.5.2.3.7- preconditions 30
3.5.2.4- Rubriques concernant la validation de la spécification 30
3.5.2.4.1- var_test 31
3.5.2.4.2- test 31
3.5.2.4.3- var_theo 31
3.5.2.4.4- schema_induc 31
3.5.2.4.5- theoremes 33
3.5.3- Description d'une représentation 33
3.5.3.1- Exemple 33
3.5.3.2- Rubriques concernant la représentation 36
3.5.3.2.1- entête de la représentation 36
3.5.3.2.2- utilise 36
3.5.3.2.3- fonction_rep 36
3.5.3.2.4- var_rep 36
3.5.3.2.5- rep_ops 37
3.5.3.2.6- preconditions 37
3.5.3.3- Rubriques concernant la validation de la représentation 37
3.5.3.3.1- gen_var_preuve 37
3.5.3.3.2- var_theo 38
3.5.3.3.3- schema_induc 38
3.5.3.3.4- theoremes 38
3.6- Bibliothèque 38
3.7- Démonstrateur 39
3.7.1- Généralités 39
3.7.2- La réécriture 41
3.7.3- Le raisonnement par induction 42
3.7.4- Le raisonnement par cas 43
3.8- Compléteur 44

4- LPG 47

4.1- Introduction 47
4.2- Généralités 48
4.3- Généricité 50
4.4- Syntaxe 52
4.5- Les propriétés 54
4.6- Les types 58
4.6.1- Les types non génériques 58
4.6.2- Les types génériques 59
4.6.2.1- Sémantique 59
4.6.2.2- Description 62
4.7- Les enrichissements 64
4.7.1- Sémantique 64
4.7.2- Description 66
4.8- Bibliothèque prédéfinie 67
4.9- Liste des commandes 68

5- Relations sémantiques 69

5.1- Introduction 69
5.2- Importation 71
5.3- Paramétrisation 74

5.3.1- Cas général 74
5.3.2- Importation d'un module paramétré 77
5.4- Déclaration de modèles 80
5.5- Mécanisme d'instanciation 82
5.6- Clause satisfies 85
5.7- Clause inherits 87
5.8- Clause combines 89

6- Interface 93

6.1- Introduction 93
6.2- Notion de validation sémantique 94
6.2.1- Morphismes valides par définition 94
6.2.2- Morphismes à vérifier 95
6.2.3- Autres morphismes 96
6.2.4- Processus de validation 96
6.3- Les théories structurées 96
6.3.1- L'atelier LPG 96
6.3.2- Le système LCF 98
6.3.3- Comparaison des différentes approches 109
6.4- Présentation étendue 110
6.5- Les théorèmes 111
6.6- Le logiciel 112
6.6.1- Les possibilités d'interfaçage 112
6.6.2- Principe 115
6.6.3- Restrictions 116
6.6.4- Création de l'ensemble des théorèmes 117
6.6.5- Construction de la présentation étendue 118
6.6.6- Structure du logiciel 118
6.6.6.1- Principe 118
6.6.6.2- Génération de OASIS 119
6.6.6.3- L'interface 120
6.6.7- Les commandes 123
6.6.7.1- Introduction 123
6.6.7.2- Gestion de l'arborescence 124
6.6.7.3- Visualisation et déplacement dans l'arborescence 124
6.6.7.4- Gestion de l'environnement de démonstration 125
6.6.7.5- Aides en ligne 125
6.6.7.6- Divers 126
6.7- Exemple d'utilisation du logiciel 127
6.7.1- Exemple de preuve 127
6.7.2- Exemple de validation 131

7- Vers une algèbre de preuves 143

7.1- Introduction 143
7.2- Motivations 143
7.3- Manipulation des preuves en LCF 151
7.4- Proposition pour LPG 152
7.4.1- Introduction 152
7.4.2- Hypothèses réalisées 153

7.4.3- Sortes manipulées 153
7.4.4- Principales fonctionnalités 158
7.4.5- Schéma de l'atelier de démonstration 160
7.4.6- Autres fonctionnalités 163
7.4.7- Commandes du démonstrateur 166
7.4.8- Exemple d'extraction de preuve 166
7.5- Vers une algèbre de preuves 171

8- Conclusion 173

9- Bibliographie 177

10- Annexes 187

10.1- Contenu des fichiers OASIS 187
10.1.1- Fichier de type SE 187
10.1.2- Fichier de type SO 188
10.1.3- Fichier de type SSO 189
10.1.4- Fichier de type SPO 190
10.2- Bibliothèque du système OASIS 192
10.2.1- bool 192
10.2.2- entier 193
10.2.3- enrich_entier 193
10.2.4- couple 193
10.2.5- seq 193
10.2.6- ensemble 194
10.2.7- pile 194
10.2.8- vecteur 195
10.2.9- file 195
10.2.10- liste 195
10.2.11- groupe 195
10.2.12- graphe 195
10.3- Bibliothèque du système LPG 196
10.3.1- Les propriétés 196
10.3.2- Les types 197
10.3.3- Les enrichissements 198
10.4- Modification des identificateurs OASIS 199

Liste des Figures

Fonctionnalités du système OASIS 17
Importation d'un module paramétré 77
Relations concernant la propriété Total_Order 97
Arbre de preuve en LCF 99
Structuration des théories en LCF 101
Théorie des Listes en LCF 104
Théorie LCF utilisant le renommage 105
Théorie LCF n'utilisant pas le renommage 105
Théorie LCF des tables de symboles 107
Théorie LCF des tables de symboles avec image inverse 108
Structure générale de l'interface 121
Arbre de dépendance de la propriété MC 133
Arbre de preuve de ((a*b)*c = a*(b*c)) 141
Arbre de preuve de (1*a = a) 145
Arbre de preuve de ((a+b)+c = a+(b+c)) 145
Arbre de preuve de (a+b = b+a) 145
Arbre de preuve de (a*b = b*a) 146
Arbre de preuve de (b+0 = b) 147
Arbre de preuve de (0+a = a) 147
Arbre de preuve de (a='a = vrai) 147
Arbre de preuve de (b='c = c='b) 147
Arbre de preuve de (b='c = c='b) 148
Arbre de preuve de (a<=b ou b<=a = vrai) 148
Arbre de preuve de (d='c et c='f et non(d='f) = faux) 150
Arbre de preuve de (d<=e et e<=f et non(d<=f) = faux) 150
Schéma de l'atelier de démonstration 161
Extraction de la justification d'une démonstration 167
Diagramme syntaxique d'un identificateur de OASIS 200

1- Introduction

Un algorithme est un processus de calcul permettant d'arriver à un résultat final déterminé (définition du dictionnaire *Petit Larousse*). Des méthodes algorithmiques nous permettraient donc peut-être de résoudre des problèmes via un certain nombre de calculs.

Un problème étant posé et une méthode pour le résoudre étant connue, il est nécessaire de représenter ce problème et cette méthode en utilisant des techniques algorithmiques dans un formalisme manipulable par le calculateur pour que ce dernier puisse fournir une réponse. Il est évident que la personne qui réalisera ces représentations doit connaître parfaitement le problème initial et la méthode retenue.

On peut faire quelques remarques concernant ce cheminement vers la réponse. La compréhension du problème et de la méthode, quand elle est proposée, est propre à l'être humain. Seule la pédagogie peut éventuellement apporter une aide. Quand elle n'est pas proposée, la méthode doit être découverte dans un tout premier temps. Le passage à un algorithme s'effectue lentement (c'est la phase d'analyse du problème) et on ne connaît actuellement que peu de moyens pour s'assurer de la validité de l'algorithme (l'algorithme imaginé traduit-il exactement la méthode ?). La traduction de l'algorithme en un formalisme manipulable par la machine (traduction des algorithmes dans un langage précis, puis en langage machine) peut être faite, quant à elle, de façon relativement automatique. Certains contrôles sur la cohérence de l'algorithme sont réalisés. Ce sont les phases d'analyse lexicographique, syntaxique ou d'analyse des types, par exemple.

L'idéal serait d'être sûr (de pouvoir prouver) que le programme résultat apportera bien la réponse attendue. On peut essayer de se rapprocher de cet idéal utopique en raisonnant sur le programme obtenu. Supposant qu'un programme représente correctement un problème et une méthode de résolution particulière, si un fait est vrai relativement au problème, la représentation de ce fait doit être vraie relativement au programme.

Un moyen de formaliser un problème est d'en écrire une spécification. On peut distinguer deux types de spécifications.

Une spécification peut consister à construire un modèle du problème, c'est à dire des domaines, des fonctions et en fin de compte des algorithmes pour résoudre le problème.

Une spécification peut également consister à axiomatiser le problème, c'est à dire à donner des lois du domaine de calcul. Cette méthode n'est pas toujours efficace pour dégager un algorithme, mais elle permet de raisonner sur le problème. Une telle approche peut parfois contenir un volet description de modèle, comme c'est le cas dans le langage LPG, Langage de Programmation Générique, qui a été utilisé dans le cadre de ce travail.

Cette thèse essaie de poser une dalle sur le chemin qui se dirige vers l'idéal utopique en tentant de prouver que des spécifications sont valides dans une certaine mesure. C'est à dire qu'elles respectent bien certaines contraintes imposées par la nature même du problème.

Une étude théorique a consisté à retrouver certaines conditions qui doivent être vérifiées pour qu'une spécification soit correcte. Cette étude a été poursuivie par une implémentation utilisant le langage LPG dans lequel les spécifications sont écrites et le système OASIS, Outil d'Aide à la Spécification Intercative et Structurée, qui fournit, dans notre cadre, essentiellement un outil de démonstration semi-automatique.

Il faut remarquer qu'une partie importante de cette thèse a été consacrée à l'étude des outils (LPG et OASIS) aussi bien du point de vue de leur utilisation que du point de vue de leur fonctionnement interne. Cet effort se justifie du fait qu'il fallait réunir les deux systèmes afin de les faire fonctionner et coopérer. Ce document reflète cet effort dans la mesure où on peut y trouver une description assez détaillée de ces outils.

Dans un premier temps, on s'attachera à décrire le système OASIS et plus particulièrement la partie concernant le démonstrateur de théorèmes. Un deuxième chapitre décrira le langage LPG dans lequel seront écrites les spécifications que l'on tentera de valider sémantiquement. Une attention particulière d'un point de vue pédagogique a été portée sur la rédaction de ces deux chapitres.

On formalisera ensuite les relations structurant les spécifications écrites en LPG. Cette formalisation permettra d'exhiber des conditions sémantiques qui doivent être vérifiées pour que les spécifications soient correctes suivant la sémantique du langage. Le processus qui consiste à vérifier ces conditions sémantiques sera appelé par la suite *validation sémantique*.

Le chapitre suivant concernera la description du logiciel issu de cette étude. Une première partie de ce logiciel implémente une interface entre les deux systèmes LPG et OASIS. Elle permet, d'une part, d'échanger des informations (e.g. des formules logiques) entre les deux systèmes et, d'autre part, l'utilisation quasi transparente sous l'environnement LPG du démonstrateur de OASIS. La deuxième partie de ce logiciel implémente le processus pour retrouver les formules logiques à démontrer afin de pouvoir effectivement réaliser des validations sémantiques. On formalise dans un premier temps cette notion de validation sémantique. On explique ensuite la notion de théories structurées en comparant le système LPG avec un système construit sur LCF (Logic for Computable Functions). On présente enfin le logiciel lui-même aussi bien d'un point de vue de son implémentation que d'un point de vue de l'utilisateur. Cette partie se veut également pédagogique. On donne enfin des exemples illustrant l'utilisation de ce logiciel.

L'utilisation de ce logiciel de validation sur différents exemples pratiques nous a suggéré certaines automatisations pour conduire les preuves. Ces automatisations sont à la base de la notion d'algèbre de preuves dont nous présentons une introduction dans le dernier chapitre de cet ouvrage. On décrit à cette occasion différentes fonctionnalités que l'on estime intéressantes et qui devraient figurer dans un atelier de démonstration. Cette description est faite en utilisant le formalisme de LPG.

2- Définitions

2-1- Introduction

Etant données des spécifications écrites en LPG, le but de ce travail est de pouvoir montrer que certaines conditions intrinsèques à ces spécifications sont vérifiées. Autrement dit, il faut être capable de raisonner sur les spécifications elles-même.

Un moyen d'atteindre ce but est de formaliser ces différentes notions telles que les spécifications, les conditions sémantiques, etc... Certains outils mathématiques permettent de réaliser ces formalisations.

Nous allons donc tout d'abord donner un certain nombre de définitions qui seront employées tout au long de ce document. On présente dans un premier temps les concepts de signature et de présentation qui permettent de formaliser la syntaxe et la sémantique d'une spécification. On introduit alors les concepts de théories et de morphismes de théories qui permettent de formaliser les relations existant entre diverses spécifications et ainsi d'exhiber certaines conditions sémantiques. On formalise alors ce qu'est le processus de réécriture qui nous permet de faire soit des calculs, soit des démonstrations. On voit enfin comment on peut utiliser ces différents concepts.

Afin d'obtenir plus de détails sur ces différents concepts ainsi que divers exemples, on peut consulter divers ouvrages tels que [BG 77], [EFH 83], [EM 85], [FGMO 87], [GB 84], [GH 78], [GHM 78], [Gut 77], [Klo 87], [Pad 84].

2-2- Signatures et présentations

Une **signature** $\Sigma=(S,\Omega)$ est constituée de :

- S : un ensemble de symboles appelés **sortes**,
- Ω : une famille d'ensembles de symboles appelés **opérateurs** indexés par $S^*\times S$. Ces ensembles peuvent se diviser en K_s, ensembles de symboles indexés par S (les constantes) et $\Omega_{s_1,\ldots,s_n,s}$, ensembles de symboles indexés par $S^+\times S$.

T_Σ dénote l'ensemble des termes qui sont construits en utilisant les opérateurs de Σ (T_Σ est encore appelé algèbre des termes).

$T_\Sigma(X)$ dénote l'ensemble des termes dans lesquels il peut y avoir des variables (algèbre libre sur un ensemble X de variables).

On note $\mathcal{V}(t)$ l'ensemble des variables d'un terme t donné. Si $\mathcal{V}(t)=\varnothing$, on dira que le **terme** t est **fermé**.

Une **algèbre** $A=(S_A,\Omega_A)$ relative à une signature $\Sigma=(S,\Omega)$, encore appelée Σ-algèbre est constituée de deux familles :

- $S_A=(A_s)_{s\in S}$ où A_s sont des ensembles pour chaque $s\in S$,

- $\Omega_A=(F_A)_{\omega\in\Omega}$ où

  $F_A\in A_s$ pour toute constante $\omega\in K_s$ et

  $F_A : A_{s_1} \times A_{s_2} \times\ldots\times A_{s_n} \rightarrow A_s$ pour tout opérateur $\omega\in\Omega_{s_1,\ldots,s_n,s}$.

Soient deux algèbres A et B d'une même signature $\Sigma=(S,\Omega)$, un homomorphisme $f:A\rightarrow B$, encore appelé **Σ-homomorphisme**, est une famille de fonctions $f_s:A_s\rightarrow B_s$ pour tout $s\in S$ telles que

- pour chaque constante $K:\rightarrow s\in\Omega$ et $s\in S$, on a $f_s(K_A) = K_B$ et
- pour chaque symbole d'opération $\omega:s_1, \ldots, s_n\rightarrow s\in\Omega$ et pour tout $t_i\in A_{s_i}$ avec $i=1, \ldots, n$, on a $f_s(\omega_A(t_1, \ldots, t_n)) = \omega_B(f_{s_1}(t_1), \ldots, f_{s_n}(t_n))$.

Un homomorphisme $f:A\rightarrow B$ est dit **isomorphisme**, ce que l'on note $f:A\simeq B$, si toutes les fonctions $f_s:A_s\rightarrow B_s$ pour chaque $s\in S$ sont bijectives.

Les **algèbres** A et B seront dites **isomorphes**, ce que l'on note $A\simeq B$, si il existe un isomorphisme $f:A\simeq B$ ou $g:B\simeq A$.

Une **algèbre** I sera dite **initiale** dans une classe C de Σ-algèbres si

- $I\in C$ et
- pour chaque Σ-algèbre $A\in C$, il y a un unique Σ-homomorphisme $f:I\rightarrow A$ qui est appelé alors l'homomorphisme initial sur A.

Une **présentation** $P=(\Sigma,E)$ est un couple où $\Sigma=(S,\Omega)$ est une signature et E est un ensemble d'équations e notées $t_1==t_2$ avec $t_{i,i=1,2}\in T_\Sigma(X)$ et t_1 étant de la même sorte que t_2.

Etant donnés un ensemble de variables X et une Σ-algèbre A, on peut définir une fonction $f : X\rightarrow A$ qui attribue un élément de A à une variable de X. Cette fonction peut être étendue à : $f^\# : T_\Sigma(X)\rightarrow A$.

Les **modèles** d'une présentation (S,Ω,E) sont des algèbres hétérogènes $A=(S_A,\Omega_A)$ telles que, $\forall e\in E$, A satisfait e, ce que l'on note $A \models e$. Plus formellement, $\forall e : t_1==t_2$ où $t_1,t_2\in T_\Sigma(X)$ et sont de même sorte, l'algèbre A sera un modèle de la présentation (S,Ω,E) si $\forall f : X\rightarrow A$, $f^\#(t_1)=f^\#(t_2)$. La classe de ces algèbres est dénotée $Alg_{\Sigma,E}$.

On dira qu'une **signature** $\dot{\Sigma}=(S,\Omega)$ est **partielle** si il existe des opérateurs appartenant à Ω dont le profil utilise une ou plusieurs sortes n'apparaissant pas dans S. On peut remarquer que, étant donné une signature, on peut extraire plusieurs signatures partielles.

De même que pour les signatures, on qualifiera une **présentation** $\dot{P}=(\Sigma,E)$ de **partielle** si la signature Σ est partielle ou si il existe des équations de E qui utilisent

des opérateurs ou des sortes non définis dans Σ. A partir d'une présentation P, on peut obtenir plusieurs présentations partielles.

Etant donnée une signature $\Sigma=(S,\Omega)$, on peut distinguer certains éléments particuliers de S. On qualifiera ces éléments de **sortes formelles** de S. De même, on peut distinguer certains éléments particuliers de Ω. On dira que ces éléments sont des **opérateurs formels** de Ω. On peut alors noter $\Sigma=(<S^f,S>,<\Omega^f,\Omega>)$ où nous avons $S^f \subseteq S$ et $\Omega^f \subseteq \Omega$. On peut alors considérer $S \ominus S^f$ ou $\Omega \ominus \Omega^f$ pour ne parler que des objets (sortes ou opérateurs) effectifs (i.e. non formels).

2-3- Théories et morphismes de théories

Une **théorie** $Th=(\Sigma,E^\circ)$ est définie par une signature Σ et un ensemble d'équations E° fermé par réflexivité, symétrie, transitivité et substitutivité de l'égalité. Ainsi une présentation est à l'origine d'une théorie, mais une théorie peut être présentée de différentes manières en choisissant des ensembles d'axiomes (E) différents. Etant donné une présentation $P=(\Sigma,E)$ la théorie présentée par P est $Th_P=(\Sigma,E^\circ)$, on a $E \subseteq E^\circ$. E est un ensemble d'axiomes, E° est l'ensemble des théorèmes. Il faut remarquer que l'ensemble E° est généralement infini.

Les interprétations que l'on peut donner à une théorie $Th=(\Sigma,E^\circ)$ sont des algèbres $A=(S_A,\Omega_A)$ telles que toutes les équations $e \in E^\circ$ de la théorie Th sont vraies dans ces algèbres.

Si on considère une théorie $Th=(\Sigma,E^\circ)$ telle que toutes les équations de E° sont vraies pour toutes les algèbres A appartenant à une classe d'algèbres, une telle théorie est dite **théorie équationnelle**.

Le système logique qui permet d'engendrer E° à partir de E, dans le cas d'une théorie équationnelle, est défini par les règles suivantes (le symbole $\vdash$ représente le symbole de l'inférence; $t, t_1, t_2, \ldots, t_i, t_i' \in T_\Sigma(X)$; on note également $t[t'/x]$ pour parler du terme t dans lequel la variable x est remplacée par le terme t')

•	$\vdash e$	$(\forall e \in E)$
•	$\vdash t==t$	réflexivité
• $t_1==t_2$	$\vdash t_2==t_1$	symétrie
• $t_1==t_2,\ t_2==t_3$	$\vdash t_1==t_3$	transitivité
• $t_1==t_2$	$\vdash t_1[t_3/x]==t_2[t_3/x]$	substitutivité $(x \in X)$
• $t_1==t_1', \ldots, t_n==t_n'$ et $\omega \in \Omega_{s_1,\ldots,s_n,s}$,	$\vdash \omega(t_1, \ldots, t_n)==\omega(t_1', \ldots, t_n')$	congruence ($\forall i=1, \ldots,$

n, la sorte des termes t_i et t_i' est s_i)

Une théorie $Th=(\Sigma,E^\circ)$ telle que toutes les équations de E° sont vraies dans les algèbres générées et en particulier dans l'algèbre initiale est dite **théorie inductive**. Le terme *inductive* vient du fait que la validité de certaines équations dans cette théorie peut se prouver par induction structurelle. Il n'existe toutefois pas de

système logique complet qui permette d'atteindre toutes les équations de ce type de théorie.

Le principe de l'**induction structurelle** peut se formuler de la manière suivante. Soit p un prédicat défini pour tous les termes $t \in T_\Sigma(X)$ relatifs à une signature $\Sigma=(S,\Omega)$, on peut donc dire $\forall t \in T_\Sigma(X)$, p(t) est soit vrai, soit faux. L'assertion p(t) sera vraie pour tout $t \in T_\Sigma(X)$ si les conditions suivantes sont vérifiées

- étape de base : p(t) est vraie pour toutes les constantes de la signature et toutes les variables de X,
- pas d'induction : pour chaque terme $\omega(t_1, \ldots, t_n) \in T_\Sigma(X)$, si $p(t_1), \ldots, p(t_n)$ sont vraies, alors $p(\omega(t_1, \ldots, t_n))$ est vraie.

Nous avons parlé jusqu'à présent d'équations sous la forme d'une égalité entre termes. On peut introduire le concept de choix, le **si...alors...sinon**, dans une équation. Informellement, ce concept permet de choisir une équation plutôt qu'une autre. On peut voir deux manières d'introduire le si...alors...sinon.

On peut parler d'équations conditionnelles. Ce sont des cas particuliers de l'implication logique. Nous noterons $\lfloor e_1, e_2, \ldots, e_n \rfloor e$ pour parler de $e_1 \wedge e_2 \wedge \ldots \wedge e_n \Rightarrow e$ où e et e_i sont des équations au sens traditionnel. Un modèle $A=(S_A,\Omega_A)$ satisfait une équation conditionnelle $\lfloor e_1, e_2, \ldots, e_n \rfloor e$ si et seulement si $A \models e$ dès que $\forall i=1, \ldots, n$, $A \models e_i$. On peut citer certaines règles applicables pour les équations conditionnelles :

- $e_1, \lfloor e_1 \rfloor e_2 \vdash e_2$ (modus ponens)
- $(e_1 \vdash e_2) \vdash \lfloor e_1 \rfloor e_2$ (déduction sous hypothèse)

Une autre façon d'introduire ce concept est l'utilisation du **terme conditionnel** si <bool> alors t_1 sinon t_2. Une équation aura alors la forme suivante t_1==si b alors t_2 sinon t_3. Elle signifie

- t_1==si b alors t_2 sinon t_3

$\vdash \lfloor b==vrai \rfloor t_1==t_2, \lfloor b==faux \rfloor t_1==t_3$ (règle du si)

Etant données deux signatures, $\Sigma=(S,\Omega)$ et $\Sigma'=(S',\Omega')$, un **morphisme de signature** $\Phi:\Sigma \rightarrow \Sigma'$ est constitué de deux fonctions $\Phi=(\Phi_S: S \rightarrow S', \Phi_\Omega: \Omega \rightarrow \Omega')$ telles que, pour tout $\omega: s_1 \ldots s_n \rightarrow s$ appartenant à Ω avec $n \geq 0$, $\Phi_\Omega(\omega)$: $\Phi_S(s_1) \ldots \Phi_S(s_n) \rightarrow \Phi_S(s)$ appartient à Ω'. Etant donné un morphisme de signatures $\Phi:\Sigma \rightarrow \Sigma'$, on peut construire les deux applications suivantes :

- $\hat{\Phi}: T_\Sigma(X) \rightarrow T_{\Sigma'}(X')$ qui traduit des termes de la signature Σ en des termes de la signature Σ',
- $\bar{\Phi}: Alg_{\Sigma'} \rightarrow Alg_\Sigma$ qui permet d'obtenir une Σ-algèbre A à partir d'une Σ'-algèbre A' en oubliant les ensembles et les fonctions de A' qui n'ont pas d'image inverse par Φ dans A.

Soit $Th=(\Sigma,E)$ une théorie, un **morphisme de théorie** $\Phi:(\Sigma,E)\rightarrow(\Sigma',E')$ est un morphisme de signatures $\Phi_\Sigma:\Sigma\rightarrow\Sigma'$, tel que $\forall A'\in Alg_{\Sigma',E'}$, $\bar{\Phi}_\Sigma(A')\in Alg_{\Sigma,E}$. Une autre façon de donner cette définition est ([GB 84]) : $\forall e:t_1==t_2\in E$, $\hat{\Phi}_\Sigma(t_1)==\hat{\Phi}_\Sigma(t_2)\in E'$ ($\hat{\Phi}_\Sigma(t_1)==\hat{\Phi}_\Sigma(t_2)$ peut encore s'écrire $\Phi_E(e)$).

Etant donné un morphisme de théorie $\Phi:(\Sigma,E)\rightarrow(\Sigma',E')$, On dira que Φ est **persistent** si et seulement si $\forall A\in Alg_{\Sigma,E}$, $A\simeq\bar{\Phi}(A')$ où A' est librement engendrée sur A dans $Alg_{\Sigma',E'}$. On dira que Φ est **fortement persistent** si $A=\bar{\Phi}(A')$.

Soit $\Phi : (S_s, \Omega_s, E_s) \rightarrow(S_t, \Omega_t, E_t)$ morphisme de théorie entre une théorie source et une théorie cible. La **théorie cible** peut être **décomposée** en l'image de la théorie source et le reste. C'est à dire $S_t=\Phi_S(S_s)\oplus S'$, $\Omega_t=\Phi_\Omega(\Omega_s)\oplus\Omega'$ et $E_t=\Phi_E(E_s)\oplus E'$ où $\oplus$ est l'union disjointe sur les ensembles et $P'=(S', \Omega', E')$ est une présentation partielle.

2-4- Réécriture et formes normales

Etant donné un terme, une **occurrence** d'un sous-terme dans ce terme est une suite d'entiers qui désigne le chemin allant de la racine du terme à la racine du sous-terme considéré. Soit **t** un terme, on désigne par **O(t)** l'ensemble des occurrences de tous ses sous-termes. On peut définir O(t) par les règles :

- $\varepsilon \in O(t)$ où ε désigne la séquence vide (t est sous-terme de lui-même),
- $u \in O(t_i) \Rightarrow i.u \in O(\omega(t_1, ..., t_i, ..., t_n))$ $\forall i\in 1, ..., n$, $\forall\omega$ opérateur figurant dans le terme t.

On notera **t/u**, $u\in O(t)$, le sous-terme de t d'occurrence u. On a ainsi $t/\varepsilon=t$, $\omega(t_1, ..., t_i, ..., t_n)/i.u=t_i/u$.

Etant donnés deux termes t et t' et une occurence $u\in O(t)$, on notera $t[u \leftarrow t']$ le terme t dans lequel on remplace le sous-terme figurant à l'occurence u (t/u) par le terme t'.

Une **substitution** est un Σ-endomorphisme de $T_\Sigma(X)$. Autrement dit, une substitution fait correspondre à un terme de $T_\Sigma(X)$ un autre terme de $T_\Sigma(X)$.

Un **système de réécriture** $\mathfrak{R}$ est la donnée d'une signature Σ et d'un ensemble de règles de la forme $g\rightarrow d$ où g et d sont des éléments de $T_\Sigma(X)$ de même sorte tels que $\mathcal{V}(d)\subseteq\mathcal{V}(g)$, X étant un ensemble de variables.

Une **équation** $t_1==t_2$ avec $t_{i,i=1,2}\in T_\Sigma(X)$ et de même sorte sera dite **orientable** étant donné un ordre sur les termes » si on peut comparer t_1 et t_2. Si on a $t_1 \gg t_2$, on pourra considérer $t_1\rightarrow t_2$ comme une règle de réécriture.

Un terme **t se réduit en t'** à l'occurrence $u \in O(t)$ dans $\mathfrak{R}$ si et seulement si $\exists\ \sigma$ une substitution, $\exists$ i tel que $g_i \rightarrow d_i \in \mathfrak{R}$ et $t/u = \sigma(g_i)$ on a alors $t' = t[u \leftarrow \sigma(d_i)]$. On écrira $t \rightarrow t'$.

Une **relation de réduction** $\rightarrow_{\mathfrak{R}}$ est une relation binaire associée à un système de réécriture $\mathfrak{R}$. C'est la plus fine relation contenant tous les couples (g, d) où $g \rightarrow d \in \mathfrak{R}$ et fermée par substitution et remplacement. C'est à dire $\forall t_1, t_2, t_3 \in T_\Sigma(X)$, $t_1 \rightarrow_{\mathfrak{R}} t_2 \Rightarrow \forall \sigma$ une substitution, $\sigma(t_1) \rightarrow_{\mathfrak{R}} \sigma(t_2)$ et

$t_1 \rightarrow_{\mathfrak{R}} t_2 \Rightarrow t_3[u \leftarrow t_1] \rightarrow_{\mathfrak{R}} t_3[u \leftarrow t_2]$.

La fermeture transitive de la relation $\rightarrow_{\mathfrak{R}}$ est notée $\xrightarrow{*}$.

Un terme t est sous **forme normale** si et seulement si il ne peut plus être réduit. Si $t_1 \xrightarrow{*} t_2$ et t_2 est sous forme normale, on dit que t_2 est une forme normale de t_1.

Un système de réécriture est dit **nœthérien** (ou à terminaison finie) si et seulement si il n'existe aucun terme t duquel part une chaîne de réduction infinie.

Un système de réécriture est dit **confluent** si et seulement si $\forall t_1, t_2, t_3 \in T_\Sigma(X)$, $t_1 \xrightarrow{*} t_2$ et $t_1 \xrightarrow{*} t_3 \Rightarrow \exists t_4$, $t_2 \xrightarrow{*} t_4$ et $t_3 \xrightarrow{*} t_4$.

Un système de réécriture est dit **canonique** si et seulement si il est nœthérien et confluent.

Si R est un système de réécriture canonique, alors tout terme t a une et une seule forme normale que l'on note $t\downarrow$.

2-5- Utilisation de ces concepts

Ces définitions étant données, on peut voir très rapidement deux exemples d'utilisation de ces concepts. Dans toute la suite de ce document, un usage plus ou moins intensif de ces définitions sera fait.

Dans le premier exemple, on considère le moyen de réaliser des calculs en utilisant un système de réécriture. Dans le cas où on considère un type abstrait comme un modèle et si on veut donc réaliser des calculs, on s'intéresse alors à l'algèbre des termes. La réécriture peut être vue comme un moyen de calculer des valeurs. Une règle de réécriture $g \rightarrow d$ se lit alors de la manière suivante : la valeur du terme g (le membre gauche de la règle) est donnée par le terme d (le membre droit de cette même règle).

Le deuxième exemple présente un système de réécriture comme un moyen utilisé pour réaliser des démonstrations. Si on s'intéresse donc aux démonstrations, on essaie alors de faire des preuves dans des théories et on s'intéresse pour cela à l'algèbre initiale. Un système de réécriture canonique est alors un moyen efficace de prouver des égalités. Il peut arriver que l'on ne puisse pas orienter une équation. Il

faut alors utiliser les règles d'inférence de l'égalité (utilisation de la permutation) en sachant toutefois que cela peut poser divers problèmes notamment de terminaison.

Dans la suite de cet ouvrage, on s'attachera plus particulièrement à la notion de démonstration. Regardons, pour cela, quelles sont les techniques de démonstration que l'on pourra utiliser.

2-6- Techniques de démonstration

Deux manières différentes de réaliser des preuves sont actuellement connues en informatique.

La première forme la classe des *démonstrateurs automatiques de théorèmes*. Une stratégie générale de preuve est programmée et l'utilisateur n'intervient qu'initialement en donnant les axiomes et la formule à prouver, puis éventuellement en indiquant comment le système doit effectuer ses recherches. De temps à autres, pendant la réalisation de la preuve, il peut être amené à débloquer le système. La stratégie généralement utilisée dans cette méthode est qualifiée de *dirigée par le but* (on peut, par exemple, ajouter la négation du but à l'ensemble des axiomes et tenter de trouver une incohérence). Le système LCF [GMW 79], [Pau 83c], [Sok 83], par exemple utilise ce type de technique de démonstration.

La deuxième méthode consiste à conduire la démonstration étape par étape (méthode utilisée dans le système OASIS). A chaque étape correspond une règle d'inférence qui peut être plus ou moins compliquée. La preuve est alors qualifiée de *preuve conduite en avant*, dans la mesure où on part des axiomes et on essaie de déduire le théorème par application des règles d'inférence. Dans le même esprit, on peut imaginer une *preuve conduite en arrière*. On part du but que l'on veut démontrer et on le décompose en sous-buts par *application inverse* des règles d'inférence. On répète ce processus jusqu'à l'obtention d'égalités figurant parmi les axiomes ou étant des théorèmes déjà démontrés. OASIS ([Paul 85], [Ren 85]) utilise cette méthode de preuve conduite en arrière. Il décompose, en effet, un but en sous-buts par application des schémas d'induction (les règles d'inférence fournies par l'utilisateur), ou bien transforme une formule en une autre par substitution *d'égaux par des égaux* (utilisation de la réécriture). La preuve est considérée comme réussie dès que tous les sous-buts sont réduits à une constante *VRAI* (symbolisant le fait qu'un axiome ou un théorème déjà démontré a été obtenu).

3- OASIS

3.1- Introduction

Etant donnée une spécification, une partie de cette thèse a eu pour but d'exhiber certaines conditions sémantiques qui doivent être vérifiées. Ces conditions étant vérifiées, on pourra dire que la spécification est valide sémantiquement.

Ces conditions sémantiques revêtent l'aspect de théorèmes appartenant à la théorie associée à la spécification. Autrement dit, partant d'une spécification, on construit un ensemble de formules logiques et on veut ensuite montrer que ces formules sont en fait des théorèmes dans la théorie associée à la spécification. On peut essayer, pour ce faire, de montrer que chaque formule est une conséquence logique de la spécification.

Ce travail de formalisation a débouché sur une implémentation. Considérant un langage de spécification particulier, on a donc un ensemble d'axiomes donnés par la présentation et un ensemble de formules logiques calculé suivant certaines règles comme nous le verront ultérieurement. La tâche qui consiste alors à prouver que ces formules logiques sont des théorèmes peut se faire manuellement (c'est à dire en utilisant simplement du papier et un crayon) mais peut également être plus ou moins automatisée en utilisant un démonstrateur de théorèmes ou un outil d'aide à la démonstration automatique. Cette deuxième solution a bien sûr été choisie pour l'implémentation et l'outil employé pour faire ces démonstrations a été le démonstrateur qui est incorporé au système OASIS.

Le but de ce chapitre est donc de présenter le langage et le système OASIS dont le démonstrateur de théorèmes a été utilisé dans le cadre de la réalisation pratique.

Un effort particulier a été attaché à la conception et à la rédaction de ce chapitre de façon à le rendre aussi pédagogique que possible. Il peut de ce fait être vu comme un manuel d'utilisation du système OASIS. On peut également trouver une autre présentation de ce système dans [Paul 85].

3.2- L'atelier OASIS

OASIS est l'acronyme de "Outil d'Aide à la Spécification Interactive et Structurée". C'est avant tout un langage de spécification de logiciels développé au CNET (Centre National d'Etudes des Télécommunications) en 1983, résultat d'une étude démarrée en 1979 sur les spécifications formelles. Il a été écrit en PROLOG et fonctionne sous le système Multics, sous le système UNIX des machines SM90 et SUN3 et sous le système VMS des machines VAX.

OASIS permet de décrire des types abstraits à l'aide de spécifications algébriques. On peut aussi réaliser des contrôles sur ces spécifications et en faire des représentations.

Une spécification est dite algébrique dans le sens où elle utilise un formalisme algébrique, c'est à dire basé sur les notions de signature et d'équation.

Une spécification, comme son nom l'indique, décrit les lois que doit satisfaire un certain objet. Un objet peut être un booléen, une structure de donnée telle qu'une pile, un arbre ou encore tout autre chose, une personne, un véhicule, etc... Spécifier un type abstrait revient à décrire le profil des opérateurs mis en jeu ainsi que leur sémantique. On ne s'intéresse en aucun cas à une réalisation particulière de ce type abstrait.

Une représentation de type abstrait, par contre, s'attache aux choix de réalisation. Ces choix portent d'une part sur les structures de données manipulées, mais également sur les algorithmes employés.

Typiquement, la notion de spécification peut correspondre à la notion d'interface d'une procédure ADA[1] et la notion de représentation à celle de body d'une procédure ADA.

Il est bien évident qu'à une spécification peut correspondre toute une série de représentations. Il suffit de choisir des structures de données différentes ou d'écrire les programmes sous diverses formes. Le point à retenir est que la sémantique des opérateurs est décrite au niveau de la spécification et que le comportement de toute représentation de ces opérateurs doit obéir à cette sémantique.

On peut voir OASIS comme un atelier logiciel dans lequel on dispose d'outils permettant de travailler sur les spécifications et leurs représentations. Les spécifications et les représentations sont les deux sortes d'objets qu'OASIS peut manipuler. Les spécifications se verront également appeler *types*. On nommera, pour la suite, *objet OASIS* une spécification (ou un type) ou une représentation sans distinction.

Un objet OASIS est originellement un texte compréhensible par l'être humain, une suite de caractères ou symboles lisibles. Pour être manipulé par la machine et plus précisément par l'interpréteur PROLOG qui est à la base du fonctionnement interne de OASIS, cet objet doit être transformé, compilé. On parle alors de forme externe ou de forme interne de l'objet selon que l'on désigne le texte originel ou le résultat de sa compilation. Afin de pouvoir être utilisées ultérieurement, les formes externes et internes des objets OASIS peuvent être stockées sur des fichiers. La forme externe des objets OASIS, utilisable par l'être humain, n'apparaîtra que dans les fichiers ou à l'écran de visualisation du calculateur. La forme interne, quant à elle, figurera dans la mémoire de travail (ce qui est directement utilisable par l'interpréteur PROLOG) ou sur fichier.

Nous allons maintenant décrire d'une manière générale les différentes fonctionnalités que l'on peut trouver dans l'atelier OASIS. Chaque description sera accompagnée du nom abrégé de la commande correspondante ce qui permettra de comprendre plus aisément la figure récapitulative des fonctionnalités du système.

---

[1]Toutefois, seule la syntaxe figure dans l'interface d'une procédure ADA. La sémantique peut apparaître sous forme de commentaire et, en tout cas est donnée par le corps même de la procédure.

L'atelier dispose d'outils permettant la construction des formes internes(*C_T*, *C_R*), la compilation du contenu des fichiers contenant les formes externes (*C_T*, *C_R*)[1]. D'autres outils concernent la sauvegarde de la forme interne (*S_T*, *S_R*), telle quelle ou après décompilation (*S_V_T*, *S_V_R*), ce qui revient à sauvegarder la forme externe.

Bien évidemment, on trouve des outils permettant d'observer les objets dont on dispose, qu'ils soient sur fichier (*BIBLIO*) ou en mémoire de travail (*VISU*). Avec les outils de visualisation, on peut souhaiter disposer d'outils de modifications des objets. Ceci peut être réaliser de deux manières, soit en appelant un éditeur de texte (*MO*, *EMACS*, *QX*) qui édite le fichier contenant une forme externe, soit en allant modifier directement la forme interne se trouvant dans la mémoire de travail (*SU*, *AJ*, *RE*). Dans ce cas, la forme interne n'est pas elle-même manipulée au niveau utilisateur. On dit simplement que l'on veut supprimer, ajouter ou remplacer telle partie de tel objet. L'utilisateur écrit donc les modifications sous forme externe et ces dernières sont prises en compte par le système.

Les objets pouvant être manipulés, il faut maintenant vérifier qu'ils sont correctement écrits. La compilation de la forme externe constituait une première étape de vérification. Un autre outil permet de compléter le contrôle syntaxique des objets OASIS (*C*). Sémantiquement, d'autres outils permettent de réaliser certaines vérifications soit en évaluant par réécriture des expressions (*EVAL*, *EVAL_REP*, *TEST*, *TEST_REP*), soit en essayant de démontrer des théorèmes à l'aide d'un démonstrateur, autre outil de l'atelier (*PROUVER*, *PROUVER_REP*).

En effet, les types abstraits, en OASIS, sont spécifiés algébriquement. Le comportement (la sémantique) des opérateurs est donc décrit par des équations quantifiées universellement, c'est à dire valables pour toutes les valeurs des variables qui y figurent. On peut donc écrire des expressions à partir des opérateurs et les évaluer en considérant les équations définissant la sémantique des opérateurs utilisés comme des règles de réécriture qui seraient orientées du membre gauche de l'équation vers son membre droit.

Cette notion de système de réécriture peut introduire la nécessité d'autres outils d'analyse ou de vérification. OASIS dispose ainsi d'un algorithme (*K_B*) permettant de compléter un système de réécriture (en supposant qu'il soit à terminaison finie), bien connu sous le nom d'*algorithme de Knuth-Bendix*.

Enfin, l'atelier dispose d'une documentation en ligne. qui se présente sous la forme de différentes commandes dont le nom évoque le type d'aide attendu : *HELP_PREUVE*, *HELP_K_B*, etc...

La figure suivante résume les différentes fonctionnalités de OASIS.

C'est la forme interne qui est essentiellement manipulée par la machine. La forme externe n'apparaît sur ce schéma que pour expliquer le fonctionnement des commandes de sauvegardes sur fichier de la forme interne (*S_V_T* et *S_V_R*), sous forme externe.

---

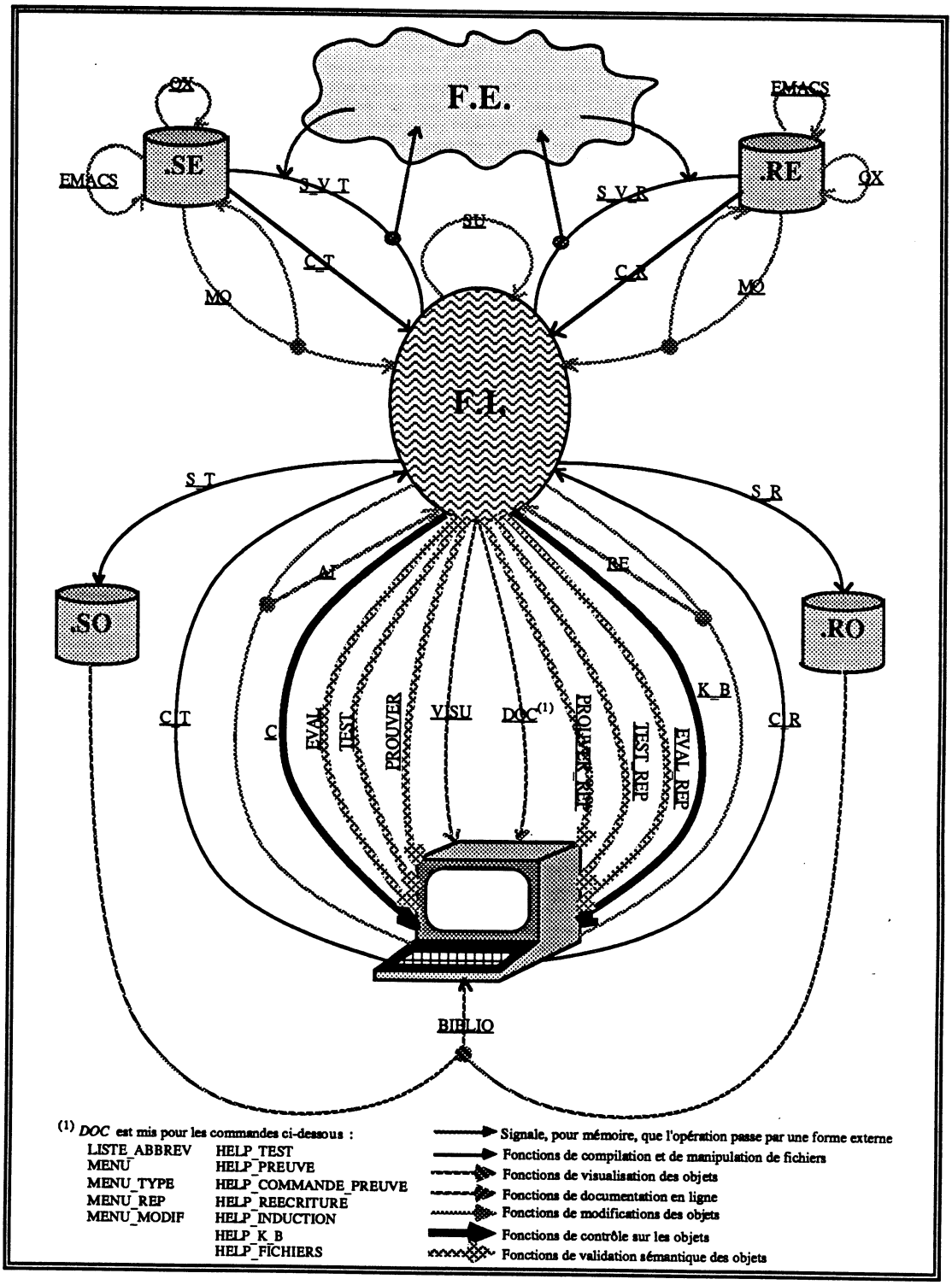
[1]Bien qu'elles aient le même nom que les commandes de construction de la forme interne, ces deux commandes sont différentes. En fait, on peut les différencier par leur nombre d'argument.

Ce schéma met en évidence le terminal (qui représente l'interface avec l'utilisateur), la forme interne et les fichiers. On peut retrouver les fonctions décrites précédemment. Elles sont divisées en six catégories :

- les fonctions de compilation et de manipulation de fichiers,
- les fonctions de visualisation des objets,
- les fonctions de documentation en ligne,
- celles de modifications des objets,
- celles de contrôle sur les objets et enfin
- celles de validation sémantique des objets.

A chaque fonction, on associe une flèche dont l'origine ainsi que son domaine d'arrivée peuvent être multiples (ex : *BIBLIO*, *MO*). Sur chaque flèche figure le nom de la fonction sous forme abrégée. Les fonctions de documentation sont regroupées sous le terme *DOC* et énumérées au bas du schéma.

F.I. désigne la forme interne et *F.E.* la forme externe. *.SE* et *.RE* se rapportent aux fichiers contenant les formes externes respectivement des spécifications et des représentations. *.SO* et *.RO* symbolisent les fichiers contenant les formes internes respectivement des spécifications et des représentations. L'icône schématise le terminal à partir duquel l'utilisateur travaille (visualisation, construction, modification, etc...).



Fonctionnalités du système OASIS

3.3- Liste des commandes possibles

Dans cette partie, nous présentons en quelque pages une synthèse des différentes commandes utilisables dans l'atelier OASIS. Sa raison d'être est qu'il est difficile de retrouver par ailleurs les informations qu'elle contient.

3.3.1- Commandes générales de l'atelier

3.3.1.1- Généralités

On donne ici la liste des fonctions générales de l'atelier qui permettent de manipuler les objets OASIS. Cette liste se trouve être divisée en six groupes conformément à la différenciation effectuée ci-dessus. On trouve le nom de la commande suivi des arguments éventuels. Une abréviation de la commande, si elle existe, est mentionnée entre parenthèse.

3.3.1.2- Compilation et manipulation de fichiers

COMPILER_REP <nom> (*C_R*) : compilation d'une représentation.

COMPILER_TYPE <nom> (*C_T*) : compilation d'un type.

CONSTRUIRE_REP (*C_R*) : construction interactive d'une représentation.

CONSTRUIRE_TYPE (*C_T*) : construction interactive d'un type.

SAUVER_REP <nom> (*S_R*) : sauvegarde de la forme interne d'une représentation.

SAUVER_TYPE <nom> (*S_T*) : sauvegarde de la forme interne d'un type.

SAUVER_VISU_REP <nom> (*S_V_R*) : sauvegarde de la forme externe d'une représentation.

SAUVER_VISU_TYPE <nom> (*S_V_T*) : sauvegarde de la forme externe d'un type.

3.3.1.3- Visualisation des objets

BIBLIO : visualisation de la bibliothèque.

VISU <nom> [<rub> [<entier$_1$> [<entier2>]]] (*VI*) : visualisation de l'objet <nom>. On peut demander à ne voir que la rubrique <rub>, soit à partie de <entier$_1$> jusqu'à la fin de la rubrique, soit de <entier$_1$> à <entier$_2$> si cette dernière valeur est mentionnée.

3.3.1.4- Modifications des objets

AJOUTER <nom> <rub> (*AJ*) : ajoute des définitions dans la rubrique <rub> de l'objet <nom>.

EMACS : appelle un éditeur pleine page du système.

MODIFIER (*MO*) : modifie l'objet courant. Cette commande se termine par une recompilation automatique de l'objet courant.

QX : appelle un éditeur ligne du système.

REMPLACER <nom> <rub> [<entier$_1$> [<entier$_2$>]] (*RE*) : cette commande est équivalente à la séquence de commandes *SUPPRIMER; AJOUTER*. Elle remplace dans la définition <nom> la rubrique <rub>. Eventuellement, on peut ne remplacer que la définition <entier$_1$> ou bien les définitions comprises entre <entier$_1$> et <entier$_2$>.

SUPPRIMER <nom> <rub> [<entier$_1$> [<entier$_2$>]] (*SU*) : supprime dans l'objet <nom> la rubrique <rub>. Il est possible, dans cette rubrique, de ne supprimer que la définition numéro <entier$_1$> ou bien les définitions comprises entre <entier$_1$> et <entier$_2$>.

3.3.1.5- Contrôles sur les objets

CONTROLE <nom$_1$> <nom$_2$> (*C*) : réalise un contrôle syntaxique de la représentation <nom$_1$> du type <nom$_2$>.

CONTROLE <nom> (*C*) : réalise un contrôle syntaxique du type <nom>.

KNUTH_BENDIX <nom> (*K_B*) : active l'algorithme de complétion sur la spécification <nom>.

3.3.1.6- Validation sémantique des objets

EVAL <nom> : réalise des évaluations interactives relatives au type <nom>. Cette commande place, en fait, l'utilisateur sous le contrôle d'un petit interpréteur.

EVAL_REP <nom$_1$> <nom$_2$> : lance une session d'évaluation relative à la représentation <nom$_1$> du type <nom$_2$>.

PROUVER <nom> <entier> : initialise une session de preuve relative au type abstrait <nom>. Le théorème à démontrer sera le théorème numéro <entier>.

PROUVER_REP <nom$_1$> <nom$_2$> <entier> : initialise une session de preuve relative au théorème numéro <entier> de la représentation <nom$_1$> du type <nom$_2$>.

TEST <nom> [<entier$_1$> [<entier$_2$>]] : réalise l'évaluation des expressions figurant dans la rubrique test du type <nom>. On peut ne demander l'évaluation que de l'expression numéro <entier$_1$>, ou bien des expressions comprises entre <entier$_1$> et <entier$_2$>.

TEST_REP <nom$_1$> <nom$_2$> [<entier$_1$> [<entier$_2$>]] : évalue les tests figurant dans la rubrique *test* de la représentation <nom$_1$> du type <nom$_2$>. De même que pour la commande *TEST*, il est possible de n'évaluer que de l'expression numéro <entier$_1$>, ou bien les expressions comprises entre <entier$_1$> et <entier$_2$>.

3.3.1.7- Documentation en ligne

HELP_COMMANDE_PREUVE : détails concernant les commandes de preuves.

HELP_FICHIERS : indication sur la gestion des fichiers en OASIS.

HELP_INDUCTION : utilisation des schémas d'induction.

HELP_K_B : utilisation de l'algorithme de complétion.

HELP_PREUVE : indications sur les fonctions de preuve.

HELP_REECRITURE : détails concernant la réécriture.

HELP_TEST : indications sur les fonctions de test.

LISTE_ABBREV : donne la liste des abréviations des commandes.

MENU : donne certaines possibilités de OASIS.

MENU_MODIF : liste des fonctions de visualisation et d'édition des objets OASIS.

MENU_REP : liste des fonctions sur les représentations.

MENU_TYPE : liste des fonctions sur les types abstraits.

3.3.1.8- Divers

-OASIS : commande inverse de la précédente (*PROLOG*) pour passer du contrôle de l'interpréteur PROLOG à celui du système OASIS.

CALCUL : permet de visualiser les étapes de réécritures réalisées lors du dernier calcul.

DETAILS_OFF : les réécritures se réalisent sans visualisation des étapes intermédiaires.

DETAILS_ON : les différentes étapes de réécriture seront montrées à l'utilisateur.

PERMUT_OFF : interdit l'utilisation des règles de réécriture permutatives.

PERMUT_ON : signale au démonstrateur qu'il peut utiliser les règles de réécritures permutatives (les règles non orientées).

PROLOG : cette commande permet de passer le contrôle à l'interpréteur PROLOG.

SAUVER_SESSION, SUSPENDRE : sauvegarde l'état courant de la session afin de la continuer ultérieurement.

3.3.2- Fonctions du démonstrateur de théorèmes

3.3.2.1- Généralités

Les fonctions présentent dans ce paragraphe concernent plus particulièrement l'outil de démonstration automatique.

3.3.2.2- Construction de l'arborescence

CAS_AUTOMATIQUE <pos> : décomposition automatique, en fonction de sa structure du nœud <pos>.

CAS_MANUEL <pos> : décomposition du nœud <pos> réalisée par l'utilisateur.

INDUCTION <pos> : déclenche le principe de l'induction sur le nœud <pos>.

REDUCAS <pos> : applique la technique de réécriture à tous les nœuds fils du nœud <pos>.

REDUCTION <pos> : applique la technique de réécriture au nœud <pos>.

3.3.2.3- Suppression dans l'arborescence

ELAGUER <pos> : au cours d'une session de preuve, permet d'effacer toute la sous-arborescence se situant sous le nœud <pos>. En particulier, cette commande, appliquée à la racine, permet de redémarrer la session de preuve courante.

3.3.2.4- Visualisation et déplacement dans l'arborescence

CAS_COURANT : visualisation de la valeur du nœud que l'on est en train de traiter.

CAS_RESTANT : liste de toutes les feuilles dont la valeur de la partie théorème est différente de la constante booléenne VRAI. Ce sont les cas qui ne sont pas encore démontrés. Lorsque la liste des cas restants est vide, le système annonce à l'utilisateur que le théorème initial est démontré sous réserve de la pertinence des cas déclarés triviaux ou impossibles.

CAS_SUPPOSE : donne la liste des cas qui ont été déclarés trivial ou impossible par l'utilisateur.

DOWN <pos> : permet de descendre dans l'arborescence sur le nœud-fils <pos>.

UP : remonte dans l'arborescence sur le nœud père.

VISU_ARBRE <pos> : visualisation de l'arborescence ou de la sous-arborescence dont <pos> est la racine.

VISU_FEUILLE : visualisation de la totalité des feuilles de l'arborescences.

VISU_NŒUD <pos> : visualisation de la valeur contenue dans le nœud <pos>.

3.3.2.5- Divers

BANALISER <pos> : commande inverse de la commande nommer, retire l'étiquette repérant le nœud <pos>.

ENREGISTRER_THEOREME <nom> <entier> : crée un lemme utilisable par le démonstrateur. Le système prend le théorème numéro <entier> de l'objet <nom> et le place en tant que règle de réécriture en lui affectant une priorité plus élevée que celle des règles traditionnelles.

IMPOSSIBLE <pos> : déclare le nœud <pos> comme étant un cas impossible. Le système va considérer que la démonstration relative au nœud <pos> est terminée.

NOMMER <pos> : permet de repérer le nœud particulier <pos> de l'arborescence en lui collant l'étiquette.

TRIVIAL <pos> : déclare le nœud <pos> comme étant un cas trivial. De même que pour la commande impossible, le système considérera comme terminée la démonstration du nœud <pos>.

3.4- Environnement

La description de l'environnement concerne les fichiers manipulés par le système et certaines caractéristiques du fonctionnement interne de OASIS comme la suspension d'une session de travail, le traitement des équations permutatives, etc...

Lorsque OASIS fonctionne, un prompt *oasis :* est affiché au terminal indiquant à l'utilisateur que les différents outils de l'atelier sont disponibles. L'activation d'un outil se fait par l'intermédiaire d'une commande écrite au terminal à l'aide, en général, des caractères majuscules de l'alphabet et est terminée par un point. Une commande peut donc s'écrire sur plusieurs lignes. Certains outils nécessitent un dialogue avec l'utilisateur qui se manifeste par des questions qui lui sont posées. Quand l'outil pose une telle question, un autre prompt est affiché au terminal >. La réponse est également écrite à l'aide des caractères majuscules de l'alphabet et doit aussi être terminée par un point.

L'atelier de spécification OASIS offre un environnement de travail bien précis dont les principales fonctionnalités ont été indiquées ci-dessus. Comme il est écrit en PROLOG, les commandes OASIS ne sont rien d'autre que des demandes de résolution de buts PROLOG. On peut donc considérer OASIS comme un niveau ajouté à l'interpréteur PROLOG. Exploitant cette caractéristique, il est possible de passer de l'environnement de OASIS à celui de PROLOG et réciproquement. Dans ce nouvel environnement, le prompt devient *?*. C'est en fait le prompt de l'interpréteur PROLOG. Cet environnement PROLOG peut être intéressant pour utiliser PROLOG en tant que tel, ou bien pour examiner de plus près ce qui se passe quant à l'utilisation de OASIS (consultation de fichiers, examen de la forme interne, etc). Le passage de OASIS à PROLOG se fait par la commande *PROLOG*. Le passage de PROLOG à OASIS se fait en résolvant le but *-OASIS*.

Comme cela a été mentionné précédemment, divers fichiers sont utilisés par cet atelier afin de stocker certaines informations. Ils peuvent être répartis en trois catégories, les fichiers concernant les spécifications, ceux relatifs aux représentations de spécifications et enfin un fichier contenant un système de réécriture issu de l'outil de complétion (algorithme de Knuth-Bendix), ce dernier fichier pouvant être considéré à part. D'une manière générale, un fichier est relatif à un objet OASIS et son nom est ainsi dérivé du nom de l'objet. Etant donné un objet de nom *object_name*, les fichiers qui pourront lui être associés auront un nom de la

forme *object_name.suffixe*. *suffixe* sera égal à *KB* pour un fichier contenant le système de réécriture complété. Pour les deux autres catégories de fichiers, *suffixe* sera une chaîne de deux ou trois caractères. Le premier de ces caractères sera *S* pour les fichiers relatifs aux spécifications et *R* pour ceux spécifiques aux représentations. Le dernier caractère sera *E* pour indiquer que le contenu est une forme externe de l'objet et *O* pour une forme interne. Eventuellement, on pourra trouver un troisième caractère entre les deux ci-dessus mentionnés. Ce sera *S* pour indiquer que le fichier contient les profils des opérateurs utilisés (la syntaxe), ou *P* repérera les fichiers contenant les règles de réécritures associées à ces opérateurs.

Etant donné, par exemple, une spécification de nom *spec_name*, nous pourront avoir les fichiers suivants :

- spec_name.SE : contient la forme externe de la spécification,
- spec_name.SO : contient le forme interne de la spécification,
- spec_name.RSO : contient l'ensemble des profils des opérateurs de la spécification, cet ensemble est obtenu par fermeture transitive des importations des spécifications,
- spec_name.RPO : contient l'ensemble, également obtenu par fermeture transitive des importations, des règles de réécriture associées aux opérateurs de la spécification,
- spec_name.KB : contient le système de réécriture obtenu par utilisation de l'outil de complétion sur la spécification.

Si on a une représentation de nom *rep_name*, nous pourrons fabriquer les fichiers :

- rep_name.RE : contient la forme externe de la représentation,
- rep_name.RO : contient la forme interne de la représentation,
- rep_name.RSO : contient l'ensemble obtenu par fermeture transitive des importations des profils des opérateurs de la représentation,
- rep_name.RPO : contient l'ensemble, également obtenu par fermeture transitive des importations, des règles de réécriture associées aux opérateurs de la représentation.

Il est possible d'avoir une idée quant à l'aspect du contenu de ces fichiers en consultant l'annexe 1.

D'autres fichiers peuvent être créés pour des besoins temporaires. Nous ne les mentionnerons pas ici.

Au cours d'une session OASIS, il peut être intéressant de récupérer une trace écrite du travail réalisé. Deux commandes OASIS sont prévues à cet effet. *SAUVER_SESSION* permet de débuter la sauvegarde de la session en cours sur fichier. *SUSPENDRE* arrête la sauvegarde de la session. Le fichier contenant le résultat de la sauvegarde n'est pas utilisable tel quel, mais doit être manipulé par des commandes particulières du système d'exploitation (fichiers *audit* du système *Multics*, par exemple).

D'autres commandes OASIS permettent de modifier l'état de fonctionnement de l'atelier en ce qui concerne essentiellement la réécriture.

Ayant réalisé une évaluation quelconque par réécriture (évaluation de test, preuve, etc...), il est possible, par la commande *CALCUL* , de visualiser la liste des différentes réécritures qui ont permis d'aboutir au dernier résultat. On peut

également activer (commande *DETAILS_ON*) ou désactiver (commande *DETAILS_OFF*) une option permettant d'obtenir une trace des réécritures réalisées au cours d'un calcul.

Nous avons vu que la sémantique des opérateurs dans le cas des spécifications algébriques est donnée par des équations. Ces équations peuvent être de deux sortes, orientables ou non orientables. Lorsqu'un opérateur est défini à l'aide d'équations orientables, le système de réécriture est capable de prendre les équations de la définition et de les utiliser comme des règles de réécriture.

Quand ces équations ne sont pas orientables, il est nécessaire de les manipuler avec précaution, notamment, de faire attention au sens dans lequel la réécriture se fera. Pour cette raison, les équations non orientables, encore appelées permutatives, présentes dans les spécifications OASIS ne sont pas exploitables par le système de réécriture. Il faut que l'utilisateur signale explicitement ce fait au système par la commande *PERMUT_ON*. Ce dernier prendra en compte les équations permutatives présentes dans les définitions en les utilisant d'une manière bien précise afin d'éviter au maximum tout problème.

Face aux problèmes théoriques posés par la prise en compte des équations non orientables, il n'y a pas de solutions générales. Pratiquement, dans OASIS ce traitement se décompose en six étapes :

- utilisation avec une priorité minimale de ces règles,
- une telle équation sera utilisée indifféremment dans les deux sens,
- après une utilisation de ce genre d'équation, on essaie de reprendre la réécriture avec celles qui sont orientables,
- tant qu'on ne peut pas simplifier l'expression courante, on les utilise en mémorisant les différentes étapes, afin d'éviter certaines boucles dans le traitement,
- dès qu'on peut simplifier l'expression courante à l'aide d'une équation orientable, on supprime les mémorisations effectuées et le processus de réécriture reprend à la première étape de ce traitement,
- le processus de réécriture s'arrêtera automatiquement dès qu'il n'est plus possible une règle permutative sans aboutir à une expression déjà obtenue.

Il est à remarquer que ce traitement particulier n'empêche nullement l'apparition de problèmes qui se manifestent généralement par des débordements de tables internes à l'interpréteur PROLOG du fait de l'exploration aveugle des différentes possibilités.

Il est possible, inversement, de demander au système de ne plus considérer les équations permutatives en utilisant la commande *PERMUT_OFF*.

Une troisième commande, *MODIF_PERMUT*, concernant l'utilisation des équations permutatives existe. Elle permet d'autoriser ou non l'utilisation du backtracking de PROLOG, d'indiquer que l'on veut lancer l'exploration en partant du sommet de l'expression ou, au contraire, en partant de l'intérieur de l'expression et, enfin, de demander ou d'annuler la mémorisation des termes irréductibles. A tout instant, l'état relatif à ces trois caractéristiques peut être consulté par la commande *ETAT_PERMUT*.

L'utilisation du backtracking de PROLOG, dans ce contexte, concerne l'exploitation des règles du type

A == si B alors C.

On réduit B, si la forme normale de B est VRAI, on transforme A en C, sinon on abandonne l'exploitation de cette règle et on en essaie une autre.

Enfin, une commande particulière, *STOP*, nous permet de quitter cet atelier de spécification.

3.5- Langage

Cette partie est consacrée à la description du langage OASIS lui-même, c'est à dire comment plus particulièrement on peut construire un type abstrait ou une représentation.

3.5.1- Syntaxe employée

La syntaxe décrite ici est la syntaxe des expressions. Ces dernières sont utilisées d'une part dans les équations définissant les opérateurs, ainsi que dans les tests et théorèmes, et d'autre part pour tester les spécifications ou représentations.

Les expressions peuvent être fonctionnelles (préfixées) ou infixées. Cette différence d'utilisation sera précisée lors de la déclaration du profil de l'opérateur. A une déclaration fonctionnelle correspondra une utilisation fonctionnelle et à une déclaration infixée correspondra une utilisation infixée. Les expressions fonctionnelles ainsi que les expressions infixées construites à partir d'opérateurs binaires seront écrites suivant la manière traditionnelle (*fact(10)*, *f(x, y)*, *5 + 8*, *elem dans ens*, etc...). Les expressions infixées construites à partir d'opérateurs n-aires (n>2) seront notées de la manière suivante (généralisation du cas des opérateurs binaires) : *graphe_initial a_un_arc_de A a B*. Dans cet exemple, l'opérateur est *a_un_arc_de...a* et a trois arguments en paramètre. Sa déclaration syntaxique (son profil) pourrait être *graphe a_un_arc_de nœud a nœud → bool* où *graphe*, *nœud* et *bool* sont des types abstraits.

Il faut remarquer qu'il n'y a pas de notion de précédence des opérateurs et que les expressions sont évaluées de gauche à droite. Une exception à cette règle est l'opérateur d'égalité qui possède une priorité inférieure à tous les autres opérateurs. Seul un parenthésage explicite peut forcer la priorité des évaluations.

Les entités lexicographiques utilisées sont les entiers naturels (nombres entiers positifs ou nuls) et les identificateurs. Les valeurs entières ont une notation traditionnelle (*0*, *58*, *61*, etc...).

Les identificateurs, quant à eux, se subdivisent en deux catégories suivant les caractères utilisés. Ils peuvent soit être écrits à l'aide des caractères majuscules et des chiffres (à condition que le premier caractère ne soit pas lui-même un chiffre) soit être écrits avec des caractères minuscules et certains autres spéciaux. En fait, tout caractère non alphanumérique différent du point, de la virgule, des parenthèses, des accolades, du blanc et du blanc souligné. Le point termine les phrases de définition du langage, la virgule et les parenthèses sont utilisées dans les notations fonctionnelles et les accolades délimitent la position de commentaires. Dans les deux catégories, on peut insérer des caractères blancs soulignés à condition qu'ils ne figurent pas en première position dans l'identificateur. On ne peut pas mélanger un formalisme avec l'autre.

Certains identificateurs sont réservés : =, *si*, *alors*, *sinon*.

3.5.2- Description d'un type

3.5.2.1- Généralités

La notion de type en OASIS permet de décrire algébriquement un type abstrait. Il faut remarquer que l'on confond le nom du type et celui de la sorte[1] déclarée dans ce type. Par exemple, la définition du type PILE déclare implicitement la sorte PILE. Il est possible d'importer des définitions d'autres types écrits par ailleurs et un type OASIS peut être défini à partir d'autres objets dont on ne connait pas la sorte à priori. Les objets (termes) de ces sortes sont des objets formels. Cette dernière notion se rapproche de celle de la généricité qui sera vue au moment de la présentation de LPG. Mise à part l'égalité, aucun opérateur n'est supposé exister sur ces objets formels.

Un type en OASIS est ainsi défini à l'aide de différentes rubriques. Il existe deux grandes catégories de rubriques, celles qui concernent la spécification du type lui-même et celles qui permettent de faire des tests de comportement relatifs à la spécification (notion de validation sémantique comme évaluation d'expressions de test, démonstration de théorèmes).

L'exemple choisi pour cette présentation est l'exemple de la spécification d'une pile. Une pile est un objet structuré qui contient des éléments (dont on ne se préoccupe pas de la nature dans cette spécification). L'ordre dans lequel sont ajoutés ces éléments est important. En effet, le dernier élément ajouté à une pile sera le premier qui pourra être retiré de cette pile[2].

Cette spécification nécessite de signaler que la pile est constituée d'objets dont on ne connait pas la sorte à priori. Pour désigner cette sorte inconnue au moment de l'écriture de la spécification, on utilise l'identificateur *elem* qui est introduit par la rubrique intitulée *parametres*. Une telle sorte (inconnue au moment de l'écriture de la spécification) est qualifiée de sorte formelle. Il est nécessaire de présenter à l'aide de la constante *PILE_VIDE* et de l'opérateur *empiler* comment une pile peut être construite. Il faut également spécifier, par les opérateurs *depiler* et *sommet* le comportement LIFO de cet objet. D'autres opérateurs sont définis sur cette structure afin de pouvoir l'observer. On a la fonction *vide* indiquant si une pile contient des éléments ou non, *taille* nous renseigne quant au nombre d'éléments de la structure et *remplacer* permet de modifier la structure.

L'utilisation de ces opérateurs d'observation nécessite la définition d'autres types abstraits, celle des *booleens* et celle des *naturels*. Ces définitions sont données dans d'autres types OASIS et sont donc importées dans cette spécification grâce à une rubrique spéciale.

On peut faire plusieurs remarques sur la structure générale de cet exemple. Tout d'abord, la définition de cet objet se termine par un point. Chaque rubrique

---

[1]On emploiera par la suite le terme *sorte* d'un objet pour désigner le type, au sens PASCAL du terme, de cet objet.

[2]Définition d'une structure de type *LIFO*, Last In First Out.

composant le type commence par un mot-clé et se termine par un point. Enfin, chaque définition à l'intérieur d'une rubrique se termine également par un point.

Voici maintenant la spécification du type des piles. Les diverses rubriques figurant dans cette spécification sont ensuite commentées et expliquées.

3.5.2.2- Exemple de la pile

```
type PILE =
parametres
    ELEM.
import
    BOOL,ENTIER.
constantes
    PILE_VIDE : PILE(ELEM).
    .
operations
    empiler(ELEM,PILE(ELEM)) -> PILE(ELEM).
    depiler(PILE(ELEM)) -> PILE(ELEM).
    sommet(PILE(ELEM)) -> ELEM.
    vide(PILE(ELEM)) -> BOOL.
    taille(PILE(ELEM)) -> ENTIER.
    remplacer(ELEM,PILE(ELEM)) -> PILE(ELEM).
    .
var_eqs
    EL : ELEM.
    P : PILE(ELEM).
    .
equations
    1 : depiler(empiler(EL,P)) == P.
    2 : sommet(empiler(EL,P)) == EL.
    3 : vide(PILE_VIDE) == VRAI.
    4 : vide(empiler(EL,P)) == FAUX.
    5 : taille(PILE_VIDE) == 0.
    6 : taille(empiler(EL,P)) == taille(P) + 1.
    7 : remplacer(EL,P) == empiler(EL,depiler(P)).
    .
preconditions
    1 : depiler(P) == si non(vide(P))
                      alors OK
                      sinon DEPILER_DE_PILE_VIDE.
    2 : sommet(P) == si non(vide(P))
                     alors OK
                     sinon SOMMET_DE_PILE_VIDE.
    3 : remplacer(EL,P) == si non(vide(P))
                           alors OK
```

```
                                    sinon REMPLACER_SUR_PILE_VIDE.
     .
var_test
     E_C : PILE(ELEM).
     EL1,EL2,EL3 : ELEM.
     .
test
     1 : sommet(depiler(empiler(EL1,empiler(EL2,empiler(EL3,PILE_VIDE))))) = EL2.
     2 : sommet(remplacer(EL2,empiler(EL1,empiler(EL3,PILE_VIDE)))) = EL2.
     3 : taille(depiler(empiler(EL3,empiler(EL1,empiler(EL2,PILE_VIDE))))) = 2.
     4 : empiler(EL2,(depiler(depiler(empiler(EL1,empiler(EL3,PILE_VIDE)))))) =
empiler(EL2,PILE_VIDE).
     .
var_theo
     U,Y,Z: PILE(ELEM).
     E,E1 : ELEM.
     .
schema_induc
     1 : P(Z) <= (P(PILE_VIDE)
                  et
                  (P(Y) => P(empiler(E,Y)))).
     .
theoremes
     1 : sommet(remplacer(E1,U)) = E1.
     2 : empiler(sommet(U),depiler(U)) = U.
     .
fin type PILE.
.
```

3.5.2.3- Rubriques concernant la spécification

Ces rubriques décrivent plus particulièrement la présentation du type abstrait.

3.5.2.3.1- parametres

Il est possible de déclarer dans cette rubrique des sortes à partir desquelles des définitions seront spécifiées. Dans notre exemple, on dit que nous allons travailler avec des piles dont les éléments seront de sorte *ELEM*. Il est possible de préciser plusieurs sortes formelles. Ces sortes paramètrisent la spécification courante.

3.5.2.3.2- import

On énumère à ce niveau les différents types existants qui seront utilisés dans notre spécification actuelle. Dans l'exemple des piles, les types BOOL et ENTIER sont utilisés pour la spécification des opérations vide et taille. Ces types sont définis par ailleurs[1] et doivent donc être importés par la spécification. Pour que la

[1]Une définition de type en OASIS ne permet de définir qu'une nouvelle sorte.

spécification soit utilisable, il faut que tous les types importés soient définis. Cette vérification est faite par la commande *CONTROLE* de OASIS.

3.5.2.3.3- constantes

Cette rubrique permet de donner les différentes valeurs constantes du type abstrait en cours de spécification. Une valeur constante pour le type des piles est *PILE_VIDE*. Dans une telle déclaration, on indique le nom de la constante ainsi que sa sorte (ici *PILE(ELEM)*). Plusieurs constantes qui auraient la même sorte peuvent être mentionnées dans la même définition.

3.5.2.3.4- operations

Une présentation de type abstrait se compose d'une signature et d'un ensemble d'équations. Une signature est formée d'un ensemble de sortes et d'une famille d'opérateurs. Dans cette rubrique, on déclare pour chaque opérateur défini sur le type abstrait sa syntaxe, c'est à dire la sorte de ses arguments et la sorte de son résultat. Six opérateurs sont ainsi définis dans notre exemple. L'opérateur remplacer prend une valeur de la sorte paramètre (*ELEM*), une valeur de sorte *PILE(ELEM)* et retourne une valeur de sorte *PILE(ELEM)*. On peut différencier l'utilisation fonctionnelle de l'utilisation infixée d'un opérateur à cet endroit. Un opérateur qui sera utilisé par notation fonctionnelle sera déclaré de la manière suivante : <op_name>(<arg_list>)→<res> (ex : *fact(entier)→entier*, pour la fonction factorielle). Une déclaration d'un opérateur utilisé de manière infixée sera : $<arg_1>$ op_1 $<arg_2>$ op_2 ... $<arg_{n-1}>$ op_n $<arg_n>$ → <res> où $<arg_i>$ désigne la sorte du $i^{ième}$ argument de l'opérateur, op_j est la $j^{ième}$ partie de l'identificateur de l'opérateur et *<res>* la sorte de son résultat.

La surcharge (attribution d'un même identificateur pour des opérations différentes) des noms des opérateurs est possible en OASIS. Il faut toutefois que le nombre ou le type des arguments des fonctions diffèrent, ce qui implique que deux constantes distinctes doivent avoir des noms différents. Il existe deux opérateurs moins - sur les entiers qui se différencient par leur nombre respectif de paramètres.

3.5.2.3.5- var_eqs

On indique ici les différentes variables (leur nom et leur sorte) qui seront utilisées dans les équations et les préconditions définissant les opérateurs de la signature.

3.5.2.3.6- equations

On écrit à cet endroit les équations qui donnent la sémantique aux opérateurs du type abstrait. Ce qui correspond à l'ensemble *E* de la présentation de ce type.

Les variables figurant dans les équations sont quantifiées universellement et doivent être déclarées dans la rubrique *var_eqs*.

Ces équations sont étiquetées soit par un numéro (valeur numérique), soit par un identificateur alphanumérique. Cette différenciation vient du fait qu'il est possible de définir des équations orientables (utilisables directement par un système de réécriture qui les considère comme des règles de réécriture orientées du membre gauche vers le membre droit de l'équation) et des équations non orientables encore

appelées permutatives (exemple de l'équation définissant la commutativité d'un opérateur). De telles équations sont repérées par des étiquettes alphanumériques et ne sont pas directement utilisables par un système de réécriture.

Le membre gauche d'une équation est une expression simple dans laquelle ne peut pas figurer d'opérateurs *si...alors* ni *si...alors...sinon*. Le membre droit par contre est une expression générale.

Dans l'exemple, les équations numéro 3 et 4 spécifient le comportement de la fonction vide dans le cas d'une pile vide de tout élément et dans le cas contraire (méthode traditionnelle de spécification).

3.5.2.3.7- preconditions

Les préconditions sont utilisées en relation avec les opérateurs partiellement définis de la signature. Elles permettent de préciser ce qui se passe justement pour les cas non définis du domaine de la fonction.

Avant toute évaluation d'un opérateur, les préconditions associées à cet opérateur, si elles existent, seront évaluées. Ce n'est que si toutes les préconditions sont satisfaites (le résultat de l'évaluation est *OK*) que l'évaluation de l'opérateur sera réalisée. Dans le cas contraire, l'évaluation courante retourne un message d'erreur précisé dans la partie *sinon* de la précondition et le calcul s'arrête.

Une précondition prend la forme d'une équation dans laquelle la racine du membre gauche est l'opérateur associé à la précondition et le membre droit est une expression conditionnelle *si...alors...sinon*. L'évaluation de l'expression formant la condition retournera la valeur vrai dans le cas où la précondition est vérifiée et faux sinon. L'expression figurant dans la partie *alors* est le terme réservé *OK*. L'expression figurant dans la partie *sinon* est le message d'erreur retourné comme résultat dans le cas où la précondition n'est pas vérifiée.

Ce choix concernant l'émission d'un message d'erreur dans le cas où une précondition n'est pas vérifiée interdit la possibilité de récupérer les cas exceptionnels et interdit également la propagation des exceptions.

Dans l'exemple des piles, il est évident que prendre la valeur du sommet d'une pile qui ne contient pas d'élément n'a aucun sens. Ce cas particulier est ainsi pris en charge par la précondition numéro 2.

3.5.2.4- Rubriques concernant la validation de la spécification

Un type abstrait étant spécifié, on peut écrire en OASIS certaines définitions pouvant servir à vérifier la cohérence de ce type. C'est la raison d'être de ces rubriques. Les définitions concernent deux catégories de vérifications possibles, celles qui vont se contenter d'évaluer des expressions dont on connait le résultat à priori, l'autre catégorie est relative à des théorèmes que l'utilisateur pourra essayer de démontrer.

On voit bien à ce niveau la différence entre l'approche calcul et l'approche preuve. Les deux rubriques (test et theoreme) sont syntaxiquement identiques. Elles ne se distinguent que par les outils mis en œuvre.

3.5.2.4.1- var_test

On déclare dans cette rubrique les variables qui seront utilisées pour l'écriture des expressions de test. De même que pour la rubrique *var_eqs*, on indique le nom et la sorte de chaque variable.

Une variable particulière, nommée *E_C*, peut être utilisée en cours d'évaluation. Le résultat de chaque évaluation est mémorisé pas le système et peut être utilisé dans une nouvelle expression en étant désigné par le nom de cette variable *E_C*. La sorte associée à cette variable sera une sorte bien précise et la variable ne pourra par conséquent que mémoriser des expressions dont la sorte correspond à la sorte indiquée dans sa définition.

Dans l'exemple, on signale que la variable *E_C* a pour sorte *PILE(ELEM)*. Chaque fois que la sorte d'une expression évaluée sera *PILE(ELEM)*, le résultat de cette évaluation sera donc mémorisé dans cette variable.

3.5.2.4.2- test

On donne une liste d'expressions[1] dont on connait le résultat à priori, utilisant éventuellement des variables déclarées dans la rubrique *var_test*. Ces expressions pourront être évaluées par la suite par la commande OASIS *EVAL* qui utilisera les équations de la spécification comme des règles de réécriture. Cette notion correspond à la notion de *jeu de test* pour les programmes algorithmiques traditionnels.

Considérons le test numéro 2 de l'exemple. Etant donné une pile contenant deux éléments, si on remplace l'élément figurant au sommet par un troisième, la valeur du sommet de la pile sera la même que ce troisième élément. Selon la sémantique intrinsèque des fonctions *remplacer*, *empiler*, *sommet* et *pile_vide*, les équations définissant ces opérateurs devront être telles que ce test soit toujours évalué à vrai.

3.5.2.4.3- var_theo

Cette rubrique permet de déclarer les différentes variables qui seront utilisées dans l'écriture des théorèmes et des schémas d'induction.

3.5.2.4.4- schema_induc

Un schéma d'induction en OASIS est une règle d'inférence qui servira pour réaliser des preuves par induction lors d'une session de démonstration. Un schéma d'induction a la forme suivante : $P(x_1, x_2, ..., x_n) <= E$ où

- *E* Est une expression de type booléen, pouvant contenir des sous-expressions de la forme $P(t_1, ..., t_n)$, les t_i étant des expressions quelconques,
- *P* est un symbole représentant une proposition quelconque qui sera instanciée par un théorème particulier,

---

[1]Dans les exemples de tests présentés dans ce document, nous sommes en présence d'égalités syntaxiques entre des termes de même sorte. Un autre exemple de test concernant la spécification des booléens peut être *non(VRAI) <=> FAUX*. *VRAI* et *FAUX* sont les deux constantes habituelles, *non* est l'opérateur de négation et <=> est l'équivalence.

• les x_i sont des variables : la proposition P sera démontrée quelles que soient les valeurs des variables x_i si on arrive à réduire l'expression E associée à la constante *VRAI*.

Le nombre de variables de la liste $x_1, ..., x_n$ indique le nombre de variables sur lesquelles on fera l'induction lors d'une session de preuve. On peut, par exemple, faire un raisonnement inductif sur une variable de sorte *naturel* (dans un théorème du genre $0 + x == x$), ce qui revient à vérifier le théorème pour toutes les valeurs entières d'une droite. On peut faire un raisonnement inductif sur deux variables de sorte *naturel*(pour démontrer la réflexivité de l'égalité sur les naturels), ce qui revient à vérifier le théorème pour tous les points à coordonnées entières d'un plan.

On peut donc considérer un schéma d'induction $P(x_i) <= E$ comme une traduction de la règle d'inférence $\frac{E}{P(x_i)}$.

Pour des raisons propres à l'implémentation de OASIS, les variables x_i ne doivent pas apparaître dans les théorèmes à démontrer.

Dans l'exemple, le schéma (induction structurelle) traduit la règle d'inférence

$$(\forall P{:}\text{proposition})\frac{P(\text{PILE_VIDE}),\ (\forall Y,\forall E)\left[P(Y)\Rightarrow P(\text{empiler}(E,Y))\right]}{(\forall Z)P(Z)},$$

c'est à dire que si une propriété est vraie dans le cas d'une pile vide et si, supposant qu'elle soit vraie pour une pile quelconque Y, on arrive à montrer qu'elle est toujours vraie pour cette même pile Y à laquelle on a ajouté un élément E, alors on peut déduire qu'elle est vraie pour toutes les piles possibles de la spécification. Dans cet exemple, on pourra s'intéresser à des théorèmes concernant des piles de toutes sortes d'éléments. Aucune supposition n'est faite quant à la nature intrinsèque du domaine des éléments, ce qui peut être une source de problème si, par exemple, le domaine se trouve être vide.

Une autre règle d'inférence

$$(\forall P{:}\text{proposition})\frac{P(\text{empiler}(E_1,\text{PILE_VIDE})),\ (\forall Y,\forall E)\left[P(Y)\Rightarrow P(\text{empiler}(E,Y))\right]}{(\forall Z)P(Z)}$$

exprime le fait qu'une proposition P est vraie quelle que soit la pile à laquelle on l'applique pourvu que cette pile contienne au moins un élément. Vue sous forme de schéma d'induction dans le formalisme d'OASIS, on aurait : P(Z) <= (P(empiler(E_1, PILE_VIDE)) et (P(Y) => P(empiler(E,Y)))).

Dans l'exemple de la spécification de la PILE, on ne trouve qu'un seul schéma d'induction. Il est possible, bien sûr, de définir autant de schémas que l'on veut et la précédente règle d'inférence pourrait fort bien être enregistrée comme deuxième schéma d'induction de la spécification.

3.5.2.4.5- theoremes

On trouve dans cette rubrique des égalités entre expressions de même sorte. Ces égalités peuvent être vues comme des formules qui devraient être vraies vis à vis la sémantique intrinsèque des opérateurs de la spécification. C'est à dire qu'il sera possible d'essayer de les démontrer par réécriture et/ou par induction en cours de session de démonstration en utilisant les équations de la spécification comme des règles de réécriture. Ces égalités sont considérées comme des conséquences logiques de la présentation du type abstrait, ce sont des théorèmes appartenant à la théorie associée.

La différence existant entre cette rubrique et la rubrique de test réside dans le fait que les tests sont des expressions de sorte quelconque qui ne s'évalueront que par réécriture. Le résultat de leur évaluation sera comparé par l'utilisateur avec le résultat auquel il s'attend. Un théorème, par contre, sera fourni en donnée au démonstrateur qui pourra le décomposer en sous-buts, prendre en compte des hypothèses, appliquer la réécriture, etc... Les théorèmes pourront donc être des expressions plus générales que les tests.

Dans l'exemple, le théorème numéro 1 ressemble à l'expression de test numéro 2. Alors que le test vérifie un cas particulier (on précise complètement la pile sur laquelle on réalise l'opération), le théorème dès qu'il sera démontré dira que cette opération est vraie pour toutes les piles possibles, exception faite pour la pile vide[1]. Le cas particulier du test se trouve ainsi généralisé.

3.5.3- Description d'une représentation

Nous allons donner maintenant l'exemple de la représentation des *piles* en utilisant les *booléens*, les *entiers* et une spécification des *vecteurs*.

Nous donnons tout d'abord la spécification du type VECTEUR. On trouve ensuite la représentation PILEVECT suivie des commentaires et explications de chacune des rubriques figurant dans cette représentation.

3.5.3.1- Exemple

L'exemple de représentation choisi sera destiné à la pile qui a été spécifiée ci-dessus. Cette représentation sera réalisée à l'aide d'un vecteur dont il faut donner la spécification.

```
type VECTEUR =
parametres
    ELEM, INDICE.
import
    BOOL.
constantes
    VECTEUR_VIDE : VECTEUR(ELEM, INDICE).
    .
```

[1]Un tel théorème pourrait se démontrer à l'aide de la deuxième règle d'inférence proposée ci-dessus.

```
operations
    modifier(VECTEUR(ELEM,INDICE),INDICE,ELEM) ->
            VECTEUR(ELEM,INDICE).
    valeur(VECTEUR(ELEM,INDICE),INDICE)  ->  ELEM.
    .
var_eqs
    EL : ELEM.
    I1,I2 : INDICE.
    V : VECTEUR(ELEM,INDICE).
    .
equations
    1 : valeur(modifier(V,I1,EL),I2) ==
                    si I1 = I2
                    alors EL
                    sinon valeur(V,I2).
    .
fin type VECTEUR.
.
```

Cette spécification prend comme paramètre deux sortes formelles, d'une part la sorte des éléments eux-mêmes et d'autre part la sorte des indices. On pourra donc utiliser des vecteurs de n'importe quoi, indicés par n'importe quoi (une caractéristique des sortes formelles est qu'on suppose toujours l'existence de l'opérateur d'égalité sur une telle sorte, comme on peut le voir dans l'équation numéro 1 (*I1 = I2*). Le type abstrait des booléens est importé du fait de l'équation conditionnelle. Une constante, *VECTEUR_VIDE*, et l'opérateur *modifier* permettent de construire des vecteurs. L'opérateur *valeur* permet de retrouver la valeur à un indice précis du vecteur, ce qui est exprimé par l'équation de la spécification.

Voici maintenant une représentation possible de la pile qui utilise le type vecteur, mais également les types des booléens et des entiers. Ces deux derniers types ne seront pas présentés ici.

On peut remarquer que le nom d'un tel objet OASIS (*PILEVECT*) est accompagné du nom du type qu'il représente (*PILE*).

```
representation PILEVECT de PILE =
utilise
    BOOL,ENTIER,VECTEUR.
fonction_rep
    X(VECTEUR(ELEM,ENTIER),ENTIER) -> PILE(ELEM).
var_rep
    EL : ELEM.
    N,I,I2 : ENTIER.
    V,VEC,VEC2 : VECTEUR(ELEM,ENTIER).
    .
rep_ops
    1 : PILE_VIDE == X(VECTEUR_VIDE,0).
```

```
    2 : empiler(EL,X(VEC,I)) == X(modifier(VEC,I+1,EL),I+1).
    3 : depiler(X(VEC,I)) == X(VEC,I - 1).
    4 : vide(X(VEC,I)) == I = 0.
    5 : sommet(X(VEC,I)) == valeur(VEC,I).
    6 : taille(X(VEC,I)) == I.
    7 : remplacer(EL,X(VEC,I)) == X(modifier(VEC,I,EL),I).
    8 : X(VEC,I) = X(VEC2,I2) ==
                    si non(I = I2)
                    alors FAUX
                    sinon si I = 0
                          alors VRAI
                          sinon
                             (valeur(VEC,I) = valeur(VEC2,I2))
                            et
                             (X(VEC,I - 1) = X(VEC2,I2 - 1)).
    .
preconditions
    1 : depiler(X(VEC,I)) ==
                    si non(I = 0)
                    alors OK
                    sinon DEPILER_DE_PILE_VIDE.
    2 : sommet(X(VEC,I)) ==
                    si non(I = 0)
                    alors OK
                    sinon SOMMET_DE_PILE_VIDE.
    3 : remplacer(EL,X(VEC,I)) ==
                    si non(I = 0)
                    alors OK
                    sinon REMPLACER_SUR_PILE_VIDE.
    .
gen_var_preuve
    P -> X(V,N).
    .
var_theo
    U,Y,Z : PILE(ELEM).
    EL,EL1 : ELEM.
    VEC : VECTEUR(ELEM,ENTIER).
    I,I2 : ENTIER.
    .
schema_induc
    1 : P(Z) <= (P(PILE_VIDE)
                 et
                   (P(Y)
                   =>
                     P(empiler(EL,Y)))
```

```
                    et
                      (non(vide(Y)) et P(Y)
                     =>
                      P(depiler(Y)))).
    .
theoremes
    1 : taille(U) >= 0.
    2 : I2 > I => (X(modifier(VEC,I2,EL1),I) = X(VEC,I)).
    .
fin representation PILEVECT.
.
```

3.5.3.2- Rubriques concernant la représentation

3.5.3.2.1- entête de la représentation

Dans l'entête d'une définition de représentation, on trouve le nom de la représentation, PILEVECT, et le nom de la spécification représentée, PILE. On rend visible ainsi à l'intérieur de la représentation les objets déclarés dans la spécification, aussi bien les opérateurs (e.g. empiler, depiler, etc), que la sorte[1] (e.g. PILE) mais également les paramètres de la spécification (e.g. ELEM).

3.5.3.2.2- utilise

La représentation d'un type abstrait va se faire à l'aide d'autres types précédemment spécifiés. Il est donc nécessaire dans une représentation de mentionner les types qui seront utilisés. Par fermeture transitive, toutes les définitions présentes dans les types importés seront disponibles pour la représentation. Dans l'exemple, on voit que les types des booléens, des entiers et des vecteurs seront utilisés.

3.5.3.2.3- fonction_rep

Cette rubrique permet d'établir la correspondance entre le type représenté et le (les) type (s) utilisé (s). Cette correspondance est formalisée par une fonction de représentation (*X*) qui applique l'espace produit cartésien des types utilisés (*VECTEUR(ELEM,ENTIER), ENTIER*) dans l'espace des valeurs du type représenté (*PILE(ELEM)*).

Informellement, une pile sera représentée par un produit cartésien à deux éléments dont le premier est un vecteur et le deuxième est l'indice du sommet (ce qui correspond à la taille de la pile).

3.5.3.2.4- var_rep

Des variables seront utilisées dans les rubriques des représentations des équations (*rep_ops*), des préconditions (*preconditions*) et des générations de variables (*gen_var_preuve*). Ces différentes variables sont déclarées à ce niveau en donnant leur nom et leur sorte.

---

[1]Il faut pouvoir en effet parler de ces objets pour les représenter.

3.5.3.2.5- rep_ops

Cette rubrique permet d'indiquer comment sera interprété chaque opérateur du type représenté en utilisant les types de représentation. Cette interprétation est formalisée par des équations qui seront utilisées comme des règles de réécriture.

Dans notre exemple, sept équations précisent comment seront interprétés les sept opérateurs de la spécification de la pile. La règle numéro 1 indique qu'une pile vide est représentée par un vecteur vide et la valeur entière 0. La règle numéro 7 exprime le fait que remplacer la valeur du sommet d'une pile revient à modifier la valeur se trouvant à l'emplacement du vecteur correspondant à la taille de la pile.

L'équation numéro 8 indique comment est implémenté l'opérateur d'égalité dans l'univers de la représentation. Dans notre cas, ceci définit comment décider de l'égalité des représentations de deux piles.

3.5.3.2.6- preconditions

Comme il est possible de tenir compte de cas d'erreur dans une spécification de type abstrait, il faut savoir comment seront représentés ces cas particuliers. Ainsi, pour chaque précondition de la spécification, on indique dans cette rubrique comment l'évaluer dans la représentation.

Dans le cas de l'opérateur *depiler*, on signale que si la partie du produit cartésien représentant l'indice du sommet de la pile est nul, on est en présence d'un cas d'erreur.

3.5.3.3- Rubriques concernant la validation de la représentation

3.5.3.3.1- gen_var_preuve

Quand on représente un type abstrait à l'aide d'autres types, il faut que la représentation soit fidèle vis à vis la spécification. Sémantiquement, les objets spécifiés et représentés doivent se comporter de la même manière. Plus formellement, on peut dire que les équations de la présentation du type abstrait doivent être des théorèmes dans la théorie de la représentation. Afin de vérifier ceci, il faut traduire les équations dans l'univers de la représentation et donc traduire tout objet à représenter en un objet utilisant les types de la représentation. Cette rubrique permet d'établir cette correspondance entre les variables utilisées dans la spécification et des variables qui pourront être utilisées dans la représentation. Pour de plus amples détails sur les problèmes de représentation, on peut se référer à [GTW 78], [LZ 74].

Dans l'exemple, les équations spécifiant le comportement d'une pile sont exprimées en fonction d'une variable *P*. Dans la représentation, il faudra manipuler un vecteur et un entier, sous forme de produit cartésien. On dit donc qu'à toute variable *P* de la spécification de la pile correspondra l'expression *X(V,N)* dans l'univers de représentation où *V* est une variable de sorte *VECTEUR* et *N* est une variable de sorte *ENTIER*. Dans la représentation, on doit donc déclarer *V* et *N* parmi les autres déclarations de variables. La variable *P*, par contre, n'a pas à être déclarée puisque sa sorte est celle du type que l'on représente.

3.5.3.3.2- var_theo

Cette rubrique a la même utilité que la rubrique *var_theo* des types. Elle permet la déclaration de variables qui seront utilisées dans les schémas d'induction et dans les théorèmes. Le nom et la sorte de chaque variable manipulée doit être indiqué.

On peut remarquer, dans l'exemple, la présence des variables U,*Y* et *Z* de type *PILE(ELEM)*.

3.5.3.3.3- schema_induc

De même que pour les types, on déclare à ce niveau des schémas d'induction qui traduisent des règles d'inférence utiles pour la démonstration de théorèmes.

Dans cet exemple, un nouveau schéma a été introduit. Il correspond à la règle d'inférence

$$(\forall P{:}\text{proposition})\frac{\begin{array}{c}P(\text{PILE_VIDE}),\\ (\forall Y,\forall EL)\left[P(Y)\Rightarrow P(\text{empiler}(EL,Y))\right],\\ (\forall Y)\left[(\text{non}(\text{vide}(Y))\wedge P(Y))\Rightarrow P(\text{depiler}(Y))\right]\end{array}}{(\forall Z)P(Z)}$$

qui exprime le fait qu'une pile est construite par l'application de *empiler* ou *depiler* (dans ce dernier cas, si la pile est non vide).

3.5.3.3.4- theoremes

On retrouve la possibilité de déclarer des théorèmes. Dans une représentation, les théorèmes seront en fait des lemmes qui serviront à mener à bien les preuves de correction de la représentation. Comme dans les schémas d'induction, il est également possible, dans ces théorèmes, d'utiliser des variables dont la sorte est celle du type représenté. En cours de démonstration, de telles variables seront remplacées par les variables correspondantes dans l'univers de la représentation (utilisation de la rubrique *gen_var_preuve*).

Dans notre exemple, le théorème numéro 1 utilise une variable de sorte *PILE(ELEM)*, c'est un invariant de représentation (la taille d'une pile est forcément positive ou nulle). Suivant la définition de l'égalité entre les représentations de deux piles (équation numéro 8), on peut dire que les valeurs éventuellement stockées dans les vecteurs de représentation au-delà de l'indice représentant le sommet de pile ne sont pas significatives. Le théorème numéro 2 exprime cette constatation.

3.6- Bibliothèque

OASIS étant un langage de spécification, plusieurs types abstraits, parmi les plus utilisés, sont déjà spécifiés et fournis à l'utilisateur comme objets prédéfinis de l'atelier sous la forme de bibliothèque. L'utilisateur a toutefois la possibilité en

général de modifier les spécifications proposées[1] dans la mesure où leurs fichiers sources (*.SE*) figurent dans l'environnement.

Voici la liste des types qui sont définis dans la bibliothèque fournie avec le système. On peut trouver en annexe les signatures de chacun de ces types.

- **bool** : Spécification traditionnelle des booléens;
- **entier** : Spécification traditionnelle des valeurs entières;
- **enrich_entier** : Définition de deux nouveaux opérateurs sur les entiers;
- **couple** : Spécification du produit cartésien à deux éléments;
- **seq** : Spécification traditionnelle des séquences;
- **ensemble** : Spécification d'un ensemble d'éléments;
- **pile** : Spécification de la structure de pile (type LIFO);
- **vecteur** : Spécification de la structure de tableau;
- **file** : Spécification d'une structure de type FIFO;
- **liste** : Spécification des listes d'éléments;
- **groupe** : Spécification d'une structure de groupe;
- **graphe** : Spécification du type abstrait des graphes.

3.7- Démonstrateur

3.7.1- Généralités

Une preuve, en OASIS, est conduite lors d'une session de démonstration pendant laquelle des commandes spécifiques sont utilisées. Au cours d'une telle session, une arborescence est construite. Cette structure de données représente à tout instant l'état de la preuve.

Les nœuds sont formés d'une proposition à démontrer (à la racine de l'arborescence, la proposition est en fait le théorème à démontrer) et d'un ensemble, éventuellement vide, d'expressions booléennes. Ces expressions booléennes sont des assertions (hypothèses faites sur les valeurs des variables) dont l'effet est local au nœud où elles figurent (et aux nœuds successeurs). Pratiquement, à partir de ces expressions, le système génère des règles de réécriture (une qui suppose l'expression comme étant vraie, l'autre comme étant fausse). Ces règles seront utilisées prioritairement par rapport aux règles de réécriture traditionnelles et pour tenter de démontrer le théorème figurant au nœud où les assertions sont déclarées, ou les sous-buts qui découlent de ce théorème.

On peut représenter l'état de la preuve de la manière suivante $\{A\}\ [H \vdash F]$ où

- $\{A\}$ est l'ensemble des axiomes et
- $[H \vdash F]$ est une séquence du buts à démontrer.

Chaque but est composé de *H*, un ensemble d'hypothèses et de *F*, une formule logique à démontrer.

[1] Il peut y avoir certaines restrictions qui seront précisées au moment voulu.

La racine de cette arborescence est initialisée au départ par la commande qui débute la session de démonstration (*PROUVER*, *PROUVER_REP*) en créant un nœud contenant le théorème initial.

L'état de la preuve est alors : $\{A\}[\emptyset \vdash th]$.

{A} est initialisé avec l'ensemble des axiomes de la spécification et la séquence est initialisée avec un but à démontrer. La partie hypothèse *H* de ce but est initialement vide et la formule *F* contient la formule logique initiale que l'on veut démontrer.

La structure est ensuite développée par les différentes commandes de preuve (*REDUCTION*, *CAS_MANUEL*, *CAS_AUTOMATIQUE*, *REDUCAS*, *INDUCTION*) en créant, à partir d'un but quelconque de l'arborescence, un ou plusieurs sous-buts, jusqu'à l'obtention de la constante booléenne *VRAI* comme nouveau but. Au fur et à mesure que l'arborescence se construit, l'état de la preuve est mis à jour. Une réduction, par exemple, va simplement modifier la formule d'un but de la séquence (le but courant), une induction va créer un nouveau but dans la séquence, etc...

Un théorème figurant dans un nœud est donc impliqué par la conjonction des sous-buts se trouvant dans les nœuds immédiatement successeurs. Le théorème initial (à la racine) sera donc considéré comme démontré dès que toutes les feuilles de l'arborescence seront réduites à la constante *VRAI*[1]. Le but initial sera donc considéré comme démontré dès que tout les buts de la séquence auront la constante VRAI comme formule.

Chaque nœud de l'arborescence est étiqueté par une séquence $N_1\ N_2\ ...\ N_k$ où les N_i sont soit des nombres strictement positifs (notion d'occurrence) soit le caractère *R*. Ce caractère *R* indique que le nœud a été obtenu par simple réécriture. Une valeur numérique N_i signale que ce nœud est le $N_i^{ième}$ fils du nœud repéré par l'étiquette $N_1\ ...\ N_{i-1}$, les fils d'un nœud étant numérotés de la gauche vers la droite. Il est également possible de baptiser chaque nœud par un identificateur plus parlant.

A tout instant, l'arborescence représente l'état courant dans lequel se trouve la preuve. En cours de création, cette arborescence peut être visualisée partiellement ou non et modifiée. On peut, en effet, en détruire une partie si il s'avère que les tactiques utilisées ne mèneront pas au résultat escompté et recommencer en utilisant d'autres tactiques. *Tactique* est ici utilisé dans son sens usuel, c'est à dire une démarche suivie pour aboutir à un résultat. Les étiquettes permettent alors de désigner individuellement le nœud sur lequel on veut effectuer une opération (visualisation, destruction, etc...).

Trois techniques sont utilisées pour mener à bien une démonstration, la *réécriture*, le *raisonnement par induction* et le *raisonnement par cas*.

---

[1]Sauf cas exceptionnels, comme par exemple ceux introduits par les commandes *TRIVIAL* et *IMPOSSIBLE*.

3.7.2- La réécriture

Les équations présentes dans une spécification de type ou une représentation, ainsi que celles qui se trouvent dans les objets importés (et ceci, transitivement) constituent, pour le démonstrateur de OASIS un système de réécriture qui peut être utilisé pour réduire des termes. L'utilisateur doit s'assurer des caractères nœthérien et confluent des systèmes de réécriture utilisés.

La réécriture, en OASIS, est du type "leftmost-innermost", c'est à dire que la réécriture appliquée à un terme $t=f(t_1, t_2, ..., t_n)$, si elle ne peut pas s'appliquer au terme t lui-même, va d'abord s'appliquer au sous-terme t_1, puis aux sous-termes de t_1, et ainsi de suite en ce qui concerne t_1, puis au sous-terme t_2, etc... Les règles de réécriture sont choisies en tenant compte de leur ordre d'apparition dans les définitions.

Temporairement, il est possible d'utiliser des règles particulières d'une priorité supérieures à celle des règle traditionnelles. Ces règles particulières sont en fait les assertions que nous avons vues au paragraphe précédent, ainsi que les lemmes. En effet, seules les équations, permutatives ou non, sont prises en compte normalement pour constituer un système de réécriture. On peut toutefois demander à ce que des théorèmes existant dans une certaine définition, qu'il soient démontrés ou non, soient utilisés en tant que règle de réécriture (*ENREGISTRER_THEOREME*). Dans un tel cas, ces théorèmes sont considérés comme des lemmes et sont utilisés prioritairement par rapport aux règles de réécriture traditionnelles. Afin que la démonstration soit valable, il faudra, bien sûr, que les lemmes utilisés soient à leur tour démontrés.

L'algorithme de réécriture mémorise les différentes réécritures effectuées de façon à ne pas les refaire le cas échéant.

Des équations non orientables (permutatives) peuvent être prises en compte sur demande explicite de l'utilisateur. Quand elles ont actives, un traitement particulier est appliqué afin de réaliser la réécriture. Ce traitement a été détaillé dans la partie concernant l'environnement de OASIS.

Normalement, lorsqu'une étape de réécriture est demandée, le système utilise la technique dite des Jivaros qui consiste à réduire tout ce qu'il trouve et tant qu'il peut. Dès qu'il ne peut plus réduire quoique ce soit, il s'arrête en donnant le terme réduit. Il est possible toutefois d'obtenir des traces concernant les différentes étapes réalisées, par le positionnement d'un indicateur (*DETAILS_ON*, *DETAILS_OFF*).

On peut remarquer que le système connait la réflexivité de l'opérateur d'égalité, ainsi que l'utilisation particulière de quelques expressions :

- si <cond> alors E_1 sinon E_2 : on évalue <cond>, si elle se réécrit en vrai, on évalue E_1, si elle se réécrit en faux, on évalue E_2 sinon la réécriture s'arrête. Il faut alors générer des assertions qui supposent la valeur de la condition pour pouvoir continuer. Plus formellement, on a

- $$\frac{\text{<cond>} \xrightarrow{*} \text{VRAI}}{\text{si <cond> alors } E_1 \text{ sinon } E_2 \xrightarrow{*} E_1}$$

- $$\frac{\text{<cond>} \xrightarrow{*} \text{FAUX}}{\text{si <cond> alors } E_1 \text{ sinon } E_2 \xrightarrow{*} E_2}$$

- si on ne peut pas évaluer <cond>, la réécriture ne peut pas

s'appliquer.

• $E_1 \Rightarrow E_2$: on évalue E_1, si elle se réécrit en faux, l'expression toute entière est remplacée par vrai (on a le message *ex falso sequitur quod libet!* qui s'affiche au terminal), si elle se réécrit en vrai, on évalue E_2 sinon la réécriture s'arrête. Il faut alors générer des assertions qui supposent la valeur de l'expression E_1 pour pouvoir continuer. Plus formellement, on a

• $\dfrac{E_1 \xrightarrow{*} \text{VRAI}}{E_1 \Rightarrow E_2 \xrightarrow{*} E_2}$

• $\dfrac{E_1 \xrightarrow{*} \text{FAUX}}{E_1 \Rightarrow E_2 \xrightarrow{*} \text{VRAI}}$

• si on ne peut pas évaluer E_1, la réécriture ne peut pas s'appliquer.

Pratiquement, la réécriture s'appliquera à un nœud de l'arborescence et générera éventuellement un autre nœud (il se peut qu'aucune règle de réécriture ne puisse s'appliquer auquel cas l'arborescence n'est pas modifiée).

3.7.3- Le raisonnement par induction

Le raisonnement par induction se base sur l'utilisation de règles d'inférence qui prennent la forme de schéma d'induction en OASIS.

Une règle d'inférence peut s'écrire $\frac{E_i}{P}$ et signifie que la proposition P sera vraie si les expressions E_i sont vraies. Dans le cas du raisonnement par induction, la proposition prend en compte une ou plusieurs variables et une expression. La règle s'écrit $\frac{E}{P(X_j)}$. Ce qui veut dire que la proposition P sera vraie quelque soient les valeurs des X_j si l'expressions E est vraie. L'expression E peut contenir la proposition elle-même : $\frac{op(t_1, ..., P(e_j), ..., t_n)}{P(X_j)}$ où les t_i peuvent également contenir des expressions construites à partir de la proposition et *op* est un opérateur booléen à *n* arguments. Cela peut correspondre au fait que la proposition sera vraie pour toute valeur de X_j si on arrive à montrer qu'elle est vraie pour certaines expressions particulières. Ce type de conclusion dépend de la sémantique de l'opérateur *op*. Un cas particulier de règle d'inférence est l'induction structurelle. Supposons un domaine dont les valeurs peuvent être construites à partir d'une valeur initiale *0* et d'un opérateur *s*. L'induction consiste à vérifier la proposition pour la valeur initiale et, supposant la proposition vraie pour une valeur quelconque *y*, vérifier la proposition pour la valeur *s(y)*. La règle d'inférence correspondante est $\frac{P(0) \text{ et } (P(y) \Rightarrow P(s(y)))}{P(x)}$. On a vu dans un précédent paragraphe la correspondance entre les règles d'inférence et les schémas d'induction OASIS.

L'induction s'appliquera à un nœud particulier de l'arborescence. Suivant le schéma d'induction utilisé, un ou plusieurs nœuds seront générés correspondant aux différentes conjonctions figurant dans le schéma. Il y a donc décomposition d'un but en sous-buts.

Les variables introduites par le schéma d'induction doivent être différentes des variables du théorème. Le système ne gérant pas les noms des variables introduites, c'est à l'utilisateur de prendre les précautions nécessaires.

Il faut remarquer que cette implémentation interdit, en général, l'utilisation plusieurs fois de suite du même schéma d'induction lors de la démonstration d'un théorème. La première fois, le schéma va introduire des variables (s'il n'en introduit pas, le problème ne se pose pas). Dès la deuxième utilisation du même schéma, les mêmes variables seront de nouveau introduites et entreront en conflit avec les précédentes. On ne pourra donc utiliser plusieurs fois de suite un même schéma d'induction que si ce dernier ne contient pas de variables ou bien dans les branches générées de l'arborescence dans lesquelles il n'a pas été introduit de variables. L'utilisation plusieurs fois de suite de la règle $\frac{P(VRAI) \text{ et } P(FAUX)}{P(x)}$ pour le domaine des booléens ne posera pas de problème. La règle concernant l'induction structurelle sur les valeurs naturelles $\frac{P(0) \text{ et } (P(y) \Rightarrow P(s(y)))}{P(x)}$ posera des problèmes si elle est utilisée une autre fois dans la branche générée concernant l'implication. En effet, la variable *y* y aura été introduite par la première utilisation et cette même variable sera de nouveau utilisée par la deuxième utilisation du schéma. Une deuxième utilisation dans la branche générée pour le cas *P(0)* ne présentera pas de difficultés (*P(0)* n'introduit pas de variable).

3.7.4- Le raisonnement par cas

Le raisonnement par cas consiste décomposer un but en sous-buts en fonction de certains critères. Cette décomposition peut se faire automatiquement (*CAS_AUTOMATIQUE*) ou manuellement (*CAS_MANUEL*).

Dans le cas d'une décomposition automatique, le système analyse la structure du théorème à démontrer. Il génère un sous-but pour chaque conjonction rencontrée, ce que l'on peut écrire

$$\frac{\{A\}\ [H \vdash E_1 \wedge E_2 \wedge \ldots \wedge E_n]}{\{A\}\ [H \vdash E_1,\ H \vdash E_2\ ,\ldots,\ H \vdash E_n]}.$$

Il rend explicite les hypothèses des implications ($E_1 \Rightarrow E_2$ génère un sous-but E_2 en tenant compte de l'assertion E_1), ce qui peut se représenter par

$$\frac{\{A\}\ [H \vdash E_1 \Rightarrow E_2]}{\{A\}\ [H \cup E_1 \vdash E_2]}.$$

Enfin, il normalise les expressions constituées de *si...alors...sinon...* imbriqués. Appliqué à un nœud de l'arborescence, il y aura donc génération de un ou plusieurs fils.

Dans le cas d'une décomposition manuelle, l'utilisateur doit fournir des expressions booléennes relatives au théorème à démontrer. Ces expressions sont considérées comme des assertions et sont transformées en règles de réécriture (voir ci-dessus). Schématiquement, on peut dire qu'à une expression correspond deux

règles. Appliqué à un nœud de l'arborescence, il y aura donc génération de plusieurs sous-buts. On peut écrire

$$\frac{\{A\}\ [H \vdash t(\ldots, e, \ldots)]}{\{A\}\ [H \cup \{e{=}{=}VRAI\} \vdash t(\ldots, e, \ldots),\ H \cup \{e{=}{=}FAUX\} \vdash t(\ldots, e, \ldots)]}$$

où *e* est une expression booléenne.

Lors de la génération des règles à partir des expressions, le système tient compte de certaines propriétés d'opérateurs booléens. Il sait que l'égalité est commutative (*a=b* génère *a=b==vrai* (soit encore *a==b*) et *a=b==faux, b=a==faux*). Il tient compte de l'axiomatisation du non : *non(a)==vrai* devient *a==faux.*

Enfin, lors de la génération de règles de réécriture additionnelles lors d'un raisonnement par cas automatique ou manuel, OASIS peut également transformer l'ensemble d'assertions obtenu à l'aide de l'algorithme de Knuth-Bendix. Les performances du système en sont ainsi beaucoup améliorée. Par exemple, les deux assertions :

- f(a) == 0 et
- g(f(a)) == 1

seront transformées en :

- f(a) == 0 et
- g(0) == 1.

L'algorithme de Knuth-Bendix termine toujours dans ce cas car les variables qui apparaissent dans les assertions sont skolemisées, c'est à dire considérées comme des constantes. On sait par ailleurs que l'algorithme de Knuth-Bendix termine toujours sur un système d'équations sans variables.

3.8- Compléteur

Pour être utilisable, un système de réécriture doit être nœthérien et confluent. Dans OASIS, la vérification de la première propriété est à la charge de l'utilisateur. Un outil est par contre intégré à l'atelier permettant la vérification, dans certaines conditions, de la propriété de confluence.

Cet outil, qui n'est autre qu'une implémentation de l'algorithme de Knuth-Bendix, permet de compléter un système de réécriture pour le rendre confluent. On peut également l'utiliser afin de détecter des incohérences dans la spécification (relations non désirées entre les constructeurs) ou de faire des preuves inductives.

La version existant dans OASIS ne prend en compte ni les équations permutatives, ni les équations conditionnelles. On demande simplement d'appliquer cet algorithme sur une spécification particulière. Les équations simples de cette spécification ainsi que celles des types qui sont importés ont récupérées pour être analysées. L'utilisateur doit, par ailleurs, fournir un niveau de complexité relatif aux règles au-delà duquel une règle générée sera rejetée. Les règles générées sont orientées automatiquement du terme qui contient plus de variables vers le terme qui en contient moins. En cas d'égalité du nombre de variables, l'utilisateur doit intervenir pour l'orientation. On ne trouve pas, en effet, d'ordre prédéfini sur les termes.

Par nature, un algorithme de Knuth-Bendix peut ne pas terminer. En OASIS, en cas de terminaison, le système sauvegarde les règles produites dans un fichier particulier. Il n'y a, en effet, pas d'interaction entre l'algorithme de complétion et le reste de l'atelier. L'utilisateur peut alors utiliser le résultat contenu dans ce fichier pour éventuellement compléter sa spécification.

4- LPG

4.1- Introduction

La partie théorique de cette thèse a consisté à exhiber certaines conditions d'une spécifcication qui, si elles sont vérifiées, permettent de dire que la spécification est valide d'un point de vue sémantique. Les conditions à vérifier prennent la forme de formules logiques qui doivent être des théorèmes dans la théorie associée à la spécification. La vérification de ces conditions se traduit alors par la démonstration de ces théorèmes en considérant les axiomes de la spécification.

Cette partie théorique s'est accompagnée d'une implémentation qui a utilisé comme langage de spécification le langage LPG.

Une spécification peut être décrite en LPG par l'intermédiaire de plusieurs modules, les modules pouvant être reliés les uns aux autres par des relations explicites ou implicites comme nous le verront dans le chapitre 5.

Etant donnée une spécification écrite dans ce langage, qui se présente donc sous la forme de plusieurs modules, une partie du travail a consisté a retrouvé l'ensemble des théorèmes à démontrer.

Une deuxième partie a eu pour but de construire un ensemble d'axiomes qui sera utilisé pour démontrer les théorèmes précédement calculés.

Enfin, l'objet d'une troisième partie a été de pouvoir démontrer plus ou moins automatiquement les théorèmes sachant que le système LPG[1] ne dispose pas d'outil d'aide à la démonstration. La solution retenue a été d'utiliser un démonstrateur automatique de théorème existant, plus précisément, celui qui fait partie du système OASIS décrit dans le précédent chapitre. Cette troisième partie a consisté ainsi à relier les deux systèmes, OASIS et LPG, c'est à dire à faire en sorte qu'ils puissent s'échanger diverses informations.

Ce chapitre présente ainsi le langage et le système LPG. De même que pour le précédent chapitre, il peut être considéré comme un manuel d'utilisation de LPG. Nous avons attaché, en effet, une certaine importance au caractère pédagogique.

La version qui est décrite dans ce chapitre est une version datant de 1987 et qui fonctionne sous le système MULTICS [BDE 87]. Elle est différente de la version qui a été utilisée à l'origine dans la mesure où elle prend en compte la définition de prédicats. Les prédicats ne seront considérés que dans ce chapitre.

---

[1]LPG est à l'origine le nom d'un langage de spécification mais désigne également le système basé sur ce langage et qui offre différentes fonctionalités telles que la compilation, l'évaluation d'expression, etc...

Nous devons alors étendre la notion de signature à $\Sigma=(S,\Omega,\Pi)$ où Π est une famille d'ensembles de symboles appelés prédicats indexés par S^*. Un prédicat est défini à l'aide de clauses de Horn.

4.2- Généralités

Le nom *LPG* est issu des initiales de *Langage de Programmation Générique*. Mais cet outil est bien plus qu'un simple langage de programmation. Il est préférable de le voir comme un laboratoire de spécification dans lequel, entre autre, on peut aussi bien écrire des spécifications que des programmes.

La différence entre spécification et programme peut être très subtile.

On peut dire qu'une spécification doit permettre de décrire le comportement d'objets informatiques. Ces objets peuvent être simples (les *naturels*, les *booléens*) ou plus complexes (les *ensembles*, les *vecteurs*). Ils peuvent également être spécifiques à une application (objets représentant des *grammaires*, des *images*, etc...). Un objet informatique se compose de valeurs et d'opérateurs agissant sur ces valeurs. On peut imaginer l'exemple de l'objet informatique "pile" sur laquelle on définit des opérations comme *empiler*, *depiler*, etc... La spécification consiste, dans un langage de spécification algébrique, à exprimer des propriétés mathématiques intrinsèques à cet objet (ex : si *p* est une pile et *e* un élément de même sorte que les éléments de la pile, *depiler(empiler(e, p))==p* peut être l'expression d'une propriété des piles disant que *dépiler une pile sur laquelle on vient de mettre quelque chose donne cette même pile*).

Les objets étant spécifiés, il serait intéressant de pouvoir les manipuler en tant que tels. C'est en fait la raison d'être des programmes (ex : une pile peut être représentée à l'aide d'un tableau et d'un indice, structures de données existant dans un langage de programmation tel que PASCAL).

Un programme doit donc représenter d'une manière correcte l'objet précédemment spécifié pour l'utiliser ensuite. Etant donné la spécification d'un objet informatique, il faut être sûr que toute représentation de cet objet qui est faite par un programme traditionnel vérifie les propriétés de cet objet. C'est en fait l'un des grands intérêts des spécifications. On peut donc dire qu'un langage de spécification permet de décrire quelque chose et qu'un langage de programmation permet de représenter et d'utiliser ce quelque chose.

L'atelier LPG propose ainsi un formalisme unique, le formalisme algébrique, pour spécifier des problèmes et pour programmer des algorithmes (de manière fonctionnelle).

On entre dans cet atelier par la commande *lpg [lib_name] [options]* et on en sort par la commande *quitter*. Le prompt *Lpg :* signale le fait qu'on se trouve dans l'atelier et donc que les différents outils sont à notre disposition.

Dans cet atelier, l'utilisateur a à sa disposition un interpréteur capable d'évaluer des expressions. Il existe également un résolveur qui étant donné une liste d'expressions logiques (égalité entre termes, prédicats) peut donner les valeurs des variables (présentes dans les expressions) pour lesquelles les égalités sont vraies et les prédicats sont vérifiés. Cette caractéristique permet de voir cet outil comme un interpréteur PROLOG, en plus puissant, toutefois.

Ces possibilités d'évaluation et de résolution sont offertes dans un sous-environnement dans lequel le prompt devient *{i}* où *i* est une valeur entière indiquant le numéro du résultat de la dernière expression évaluée.

A ce niveau, il est possible soit d'évaluer des expressions fonctionnelles soit de résoudre un but (utilisation du résolveur). Ces deux outils sont totalement indépendants dans le sens où il n'y a aucune interaction entre eux (les résultats de l'un ne sont pas exploitables par l'autre et réciproquement). La détermination de l'utilisation d'un outil plutôt que l'autre se réalise à l'aide d'une séquence de caractères qui termine la commande donnée par l'utilisateur. Par conséquent, une telle commande peut s'écrire sur plusieurs lignes. Dans le cas d'une évaluation fonctionnelle, la commande se termine par la séquence ==>. Dans le cas de l'utilisation du résolveur, la commande se termine par *?*. Une commande spéciale, *fin*, permet de quitter ce sous-environnement et de se retrouver au niveau général de l'atelier.

Les spécifications et programmes sont bien entendu écrits dans un formalisme lisible par tout utilisateur et peuvent être stockés sur fichier. Le nom de ces fichiers peut être quelconque, mais suffixé par *.lpg* (ex : *predef1.lpg*). Ces fichiers doivent être compilés pour être utilisable par l'atelier. LPG dispose ainsi d'un outil de compilation réalisant une analyse syntaxique, mettant à jour diverses tables et générant un pseudo-code manipulable par l'interpréteur. Les tables contiennent l'information pertinente pour l'atelier en général.

Le contenu de ces tables peut être partiellement examiné à l'aide d'un outil de visualisation. On peut également les sauvegarder sur fichier en constituant ainsi une bibliothèque de définition et, inversement, relire les fichiers contenant les bibliothèques. Ce deuxième type de fichier est également nommé d'une manière quelconque. Ils sont toutefois suffixé par *.bib* (ex : *demo.bib*).

LPG dispose, grâce à l'implémentation résultant de cette thèse, d'un outil permettant de réaliser des démonstrations de théorèmes de façon semi-automatique. C'est en fait le démonstrateur de théorème inclus dans le système OASIS qui est utilisé. Afin de valider les spécifications, il a été décidé de choisir ce démonstrateur et, par effet de bord, il est possible de l'utiliser pour démontrer des théorèmes écrits dans les spécifications.

Mentionnons pour mémoire un dernier outil qui permet d'obtenir le contenu brut des tables de l'atelier. Cet outil est essentiellement utilisé quand on se trouve en présence de problèmes propres à LPG.

Les commandes donnant accès à ces différents outils sont écrites en caractères minuscules et possèdent des abréviations. Si l'utilisateur donne une commande non connue du système LPG, ce dernier suppose que c'est une commande du système d'exploitation et lui donne donc temporairement la main. La commande système étant achevée, LPG reprend le contrôle.

Les programmes ou spécifications, en LPG, sont construits hiérarchiquement grâce à des modules. Il y a trois espèces de modules : les *types*, les *enrichissements* et les *propriétés*. Chaque module introduit de nouvelles définitions, généralement de sortes, d'opérateurs ou de prédicats par l'intermédiaire de rubriques. La présence de chaque rubrique n'est pas obligatoire. Toutefois, l'ordre de leur apparition est important.

Il faut faire une distinction entre les *propriétés* et les autres modules. Alors que dans les *types* et les *enrichissements* on donne la sémantique des fonctions et des prédicats déclarés, dans les *propriétés* on ne fait que caractériser le comportement de classes d'objets. Autrement dit, la théorie associée à une présentation de type est la théorie inductive. La théorie associée à une présentation de propriété est la théorie équationnelle, dans le cas où cette propriété n'importe pas, ni directement, ni par l'intermédiaire d'une autre propriété, de définitions provenant de spécifications de types. Notons que la sémantique des opérateurs et des prédicats est donnée respectivement par des équations (égalité entre termes) et par des clauses de Horn.

A un instant donné, on dispose donc d'un ensemble de définitions réparties dans différents modules. Ces définitions peuvent être construites en utilisant d'autres définitions se trouvant dans d'autres modules. C'est la notion d'importation qui, en LPG, est implicite.

4.3- Généricité

La généricité est la caractéristique la plus importante et la plus originale de LPG. Cette notion signifie que des objets déclarés dans un module générique sont paramétrisés par des sortes, fonctions et prédicats.

Le plus simple est peut être de montrer tout de suite un exemple. Il est très facile, en LPG, de définir un opérateur de tri qui soit capable de fonctionner sur des listes dont on ne connait pas la nature des éléments, ni même la relation d'ordre. Il suffit de décrire le fonctionnement de cet opérateur de tri en considérant qu'il utilisera une certaine relation d'ordre formelle sur des valeurs de sorte également formelle.

L'exemple qui suit décrit un opérateur de tri dans le formalisme LPG. On suppose que la spécification des séquences (listes) est donnée par ailleurs ainsi que la définition formelle d'une relation d'ordre[1].

```
enrichment Sort                          — Specification of a sort by
                                         —   insertion function. The sort
                                         —   works on sequences. The
                                         —   elements of formal sort elem must
requires Total_Order [elem operators inf, eq] —   verify the equations of
                                         —   the total order property. inf and
                                         —   eq are the two formal operators
                                         —   which may be used in this enrichment
                                         — The specification of the
                                         —   Total_Order property is visible
                                         —   in the predef1.lpg file
                                         —   (predefined library)
operators
  sort: seq[elem] -> seq[elem]
```

[1]Le symbole -- introduit un commentaire qui se termine par la fin de ligne.

```
  add: (elem, seq[elem]) -> seq[elem]
predicates                                  — two predicates are also
  ordered: seq[elem]                        —   specified. They hold if the
  ordered2: seq[elem]                       —   parameter sequence is sorted
variables
  x,y: elem
  s: seq[elem]
equations
  1: sort(nil: seq[elem])  ==> nil: seq[elem]— nil and <+ are the two
  2: sort(x<+s) ==>add(x, sort(s))          —   constructors of the
  3: add(x, nil:seq[elem]) ==> x <+ nil:seq[elem]—   sequences
  4: add(x, y <+ s) ==> if inf (x, y)       — Notice the use of the formal
                        then x<+(y<+s)      —   operator inf as a prefixed
                        else y <+ add(x,s)—   one
                        endif
clauses
  1: ordered(s) <== sort(s) == s            — The simplest form : a sequence
                                            —   is sorted if it is the same
                                            —   as the one resulting of the
                                            —   application of the sort
                                            —   function previously defined
  2: ordered2(nil: seq[elem])               — This second predicate uses the
  3: ordered2(x <+ nil: seq[elem])          —   formal inf operator
  4: ordered2(x <+ (y <+ s)) <== inf (x, y), ordered2(y <+ s)
end Sort
```

Il est évident que pour utiliser un tel opérateur de tri générique (c'est à dire évaluer une expression LPG construite à partir de cet opérateur) il est nécessaire de donner précisément la sorte des éléments de la séquence (ex : les nombres naturels) ainsi qu'une relation d'ordre existant sur ces éléments (ex : ≥).

Un exemple d'appel à cet opérateur pourrait être *sort.Total_Order [nat* ***operators*** *>=] ([1, 3, 2])*. Dans cet exemple, *sort* est le nom de l'opérateur. *Total_Order [nat* ***operators*** *>=]*[1] indique que la relation d'ordre choisie est la fonction booléenne >= entre les naturels (qui doit être définie quelque part) et *[1, 3, 2]* est l'argument fourni à cette fonction[2]. Les symboles *[* et *]* sont utilisés en LPG pour dénoter une séquence d'éléments. Dans l'exemple, la notation *[1, 3, 2]* est équivalente à *1<+(3<+(2<+nil))* qui utilise les constructeurs (prédéfinis dans le langage LPG) des séquences (*<+* et *nil*).

Ceci est un exemple d'utilisation dans un cas particulier (en fait, une instanciation). Il est possible d'utiliser la même définition sur des séquences de

[1]Un appel à l'opérateur *sort* a en fait, en LPG, la forme suivante *sort.Total_Order [nat* ***operators*** *>=, =] ([1, 3, 2])*. L'utilité de l'opérateur = est due au fait que la fonction >= est définie en utilisant un tel opérateur d'égalité.

[2]Le résultat de l'interprétation de cette expression est *[3, 2, 1]*.

caractères, des séquences de séquences de chaînes etc... du moment qu'il existe une relation d'ordre sur les éléments de la séquence.

D'un point de vue terminologique, on dit que la fonction *sort* exige la propriété d'ordre total pour l'opérateur formel qu'il utilise. Lors d'une utilisation effective de cet opérateur de tri, il est nécessaire de lui fournir des sortes et opérateurs effectifs. On dit que ces sortes et opérateurs effectifs constituent un modèle de la propriété *Total_Order*.

En effet, une propriété décrit une classe d'objets, un modèle sera un représentant de cette classe. Pour le bon fonctionnement d'un opérateur générique exigeant la propriété *P*, il faut qu'il ait à sa disposition un modèle de *P*. Ce modèle peut soit lui être fourni lors de l'évaluation (ex : *sort.Total_Order [nat* ***operators*** *>=] ([1, 3, 2])*), c'est ce qu'on appelle l'*instanciation explicite*, soit être déjà déclaré dans un certain module. Dans ce cas, l'évaluation de l'opérateur fera référence à ce modèle à condition qu'il n'en existe qu'un seul et qu'il n'y ait pas d'ambiguïté. C'est l'*instanciation implicite* (ex : *sort ([1, 3, 2])*).

La généricité est un outil puissant qui permet de spécifier des classes d'objets. Le comportement d'un élément d'une classe est le même que celui d'un autre élément appartenant à la même classe. Bien que pouvant être utilisée dans différents cas, la spécification est ainsi écrite une seule fois, ce qui minimise les erreurs potentielles.

4.4- Syntaxe

En LPG on est amené essentiellement à écrire des expressions à partir des différents opérateurs définis. Nous allons décrire brièvement dans ce paragraphe la syntaxe utilisée pour écrire ces expressions.

Une expression peut apparaître sous trois formes différentes : les expressions conditionnelles, de liaison et les expressions fonctionnelles traditionnelles.

Il existe les expressions conditionnelles *if <cond> endif*, *if <cond> then <expr$_1$> else <expr$_2$> endif* et plus généralement *if <cond$_1$> then <expr$_1$> else_if <cond$_2$> then <exprt$_2$> else_if ... else_if <cond$_n$> then <expr$_n$> else <expr> endif*. Dans ces expressions, *cond* et *cond$_i$* sont des expressions à résultat booléen, *expr* et *expr$_i$* sont des expressions quelconques. Les deux premières sont les notations usuelles d'expressions conditionnelles. La troisième forme est un peu plus générale dans le sens où figurent plusieurs conditions et le résultat de l'expression conditionnelle est le résultat de l'expression associée à la première condition qui s'évalue à vrai. La partie *else* est optionnelle. Si aucune des conditions ne s'évalue à vrai et que cette partie existe, l'expression qui lui est associée est évaluée et si elle n'existe pas, une erreur se produit[1].

Un autre type d'expression est l'expression de liaison *let <idf$_1$> = <expr$_1$>, <idf$_2$> = <expr$_2$>, ..., <idf$_n$> = <expr$_n$> in <expr> endlet* dans laquelle *<idf$_i$>* sont des identificateurs (chaînes constituées de lettres, chiffres et _, dont la

[1]Les effets d'une telle erreur se traduisent en fait par la levée d'une exception, concept qui peut être pris en compte et donc traité par le système.

longueur est inférieure ou égale à vingt), *<expr$_i$>* sont des expressions qui peuvent se référer à *<idf$_{i-1}$>* et *<expr>* est une expression qui peut utiliser *<idf$_i$>* . Ce type d'expression permet d'évaluer une expression dans laquelle figure des identificateurs de variables qui auront été associés (liés) avant l'évaluation à d'autres expressions. Elle permet essentiellement une écriture plus lisible de l'expression *<expr>* à évaluer.

Enfin, la troisième forme d'expression est tout simplement basée sur l'utilisation d'opérateurs traditionnels ou sur des dénotations de structures prédéfinies comme les *séquences* (expressions séparées par des virgules et comprises entre les symboles *[* et *]*), les *vecteurs* (expressions séparées par des virgules et comprises entre les symboles */< et >/*), les *ensembles* (expressions séparées par des virgules et comprises entre les symboles *{* et *}*) et le *produit cartésien* (expressions séparées par des virgules ou des points-virgules et comprises entre les symboles *(* et *)*). Une *séquence* est une liste ordonnée d'éléments de même sorte d'une quelconque longueur. Un *vecteur* est une suite ordonnée d'éléments de même sorte, mais dont le nombre est fixé. Un *ensemble* est une suite non ordonnée d'éléments de même sorte, en nombre quelconque. Un *produit cartésien* est une suite d'éléments, en nombre déterminé (inférieur ou égal à vingt) et de sortes éventuellement différentes.

Les opérateurs les plus simples sont les dénotations de constantes simples (*vrai*, *faux*, constantes entières, caractères, chaînes et tout opérateur constant défini par l'utilisateur).

On trouve ensuite quelques opérateurs spéciaux comme la projection du produit cartésien qui sélectionne un composant particulier d'un produit cartésien et l'opérateur de curryfication. Informellement, l'opérateur de curryfication permet de diminuer le nombre d'argument d'autres opérateurs. Un troisième opérateur spécial est la mémoire résultat. En cours d'évaluation, les résultats des différents calculs sont mémorisés et numérotés. Un opérateur permet de retrouver ainsi un résultat précédemment calculé.

Dans le cas général, un opérateur est désigné par un symbole d'opérateur (qui est soit un identificateur, soit une chaîne d'au plus vingt caractères pris parmi *#*, *&*, ***, *+*, *-*, */*, *<*, *=*, *>*, *@*, */*, **[1], soit un symbole d'opérateur prédéfini). Ce nom peut éventuellement être suffixé par une *déclaration de modèle*. Elle permet d'indiquer quels sont les sortes, opérateurs et prédicats effectifs qu'un opérateur générique devra utiliser.

Les opérateurs s'utilisent généralement de manière préfixée. Les opérateurs binaires peuvent s'écrire en infixé ou en préfixé indifféremment. Dans le cas de l'utilisation préfixée, il peut y avoir entre un et vingt arguments, chaque argument étant lui-même une expression.

Enfin un opérateur retourne un résultat qui est d'une certaine sorte. Dans certains cas d'utilisation (afin d'éliminer des ambiguïtés), il est nécessaire de qualifier le résultat de l'opérateur en indiquant au système la sorte attendue.

---

[1] Il ne faut toutefois pas utiliser les chaînes suivantes : *->*, */<*, *>/*, *==*, *<==*, *--*, *==>* qui ont une signification spéciale pour les modules.

Quelques opérateurs particuliers (*read*, *write*, etc...) permettent de faire des entées-sorties. Ces fonctions accroissent les possibilités de l'outil de programmation.

4.5- Les propriétés

Afin d'avoir les idées claires quant à l'aspect que peut revêtir une propriété, nous présentons, à la fin de cette section, les deux squelettes possibles pour définir un tel module.

Les modules LPG *propriété* ne peuvent pas être génériques. Une propriété permet simplement la description d'objets formels (sortes, opérateurs ou prédicats). On peut donc déclarer des sortes qui ne sont rien d'autre que des identificateurs. Ces sortes modélisent le comportement de toute collection d'objets réels. On peut également déclarer des opérateurs ou des prédicats en mentionnant la sorte de leurs arguments et la sorte de leur résultat (pour les prédicats, on ne donne bien sûr que la sorte de leurs arguments). Ils pourront ou non utiliser des sortes déjà déclarées dans la propriété courante. Concrètement, ils axiomatisent tout opérateur, respectivement prédicat, effectif pourvu qu'il y ait une correspondance exacte entre les différentes sortes de leurs arguments et résultat (le nombre de paramètres de l'opérateur effectif doit être le même que le nombre de paramètres de la description formelle et il y a une unique substitution entre les sortes présentes dans le profil de la description formelle et les sortes du profil de l'opérateur effectif).

Cette partie déclaration décrit simplement la syntaxe des opérateurs. Il est possible d'écrire des équations algébriques afin de spécifier plus ou moins complètement leur sémantique. Quand un opérateur est déclaré dans une propriété, il est sensé décrire tout une classe d'opérateurs effectifs. Chacun de ces opérateurs ont des caractéristiques communes. Ces caractéristiques sont ainsi spécifiées à l'aide des équations.

Considérons l'exemple de la spécification d'une relation d'égalité. Une propriété LPG peut décrire le comportement d'un tel opérateur. Il suffit d'exprimer les propriétés (au sens usuel du terme) d'une relation d'égalité, à savoir qu'elle est réflexive, symétrique et transitive (l'utilité de la clause satisfies sera vue au chapitre concernant les relations sémantiques) :

```
property Equality
sorts t
operators
   =: (t,t)->bool                  — The abstract data type of the Boolean
variables                          —   values is defined elsewhere
   x,y,z: t
equations
  1: x=x == true
  2: x=y == y=x
  3: (x=y and y=z) => x=z == true
satisfies
  Formal_Sort[t]
end Equality
```

Plus formellement, on peut dire qu'une propriété en LPG est décrite par une présentation $P_P=(\Sigma,E)$. La sémantique d'une propriété présentée par P_P est toutes les Σ-algèbres qui satisfont E. Autrement dit, la sémantique d'une propriété est une théorie équationnelle.

Une propriété LPG peut revêtir deux formes différentes en fonction de relations qu'elle a avec d'autres propriétés (les relations entre les modules sont vues plus en détail dans le chapitre suivant). Voici successivement les squelettes à partir desquels on peut construire des propriétés.

```
property <module-name>
   sorts
      <formal-sort-name list>
   operators
      <formal-operator-declaration list>
   predicates
      <formal-predicate-declaration list>
   variables
      <variable-declaration list>
   equations
      <equation list>
   clauses
      <clause list>
   inherits
      <formal-model list>
   satisfies
      <formal-model list>
   theorems
      <theorem list>
end <module-name>
```

Cette première forme décrit l'aspect général d'une propriété, c'est à dire celui qui permet de présenter une nouvelle propriété éventuellement indépendamment de toute autre définition.

On trouve dans un premier temps une rubrique permettant de déclarer les sortes formelles (des identificateurs, ex : *elem*). Ensuite vient la rubrique de déclaration des opérateurs puis des prédicats (identifcateurs avec leur profil, ex : *point : (elem, elem) -> elem*).

On déclare alors les différentes variables (identificateurs accompagnés de leur sorte) qui seront utilisées dans les équations, clauses et théorèmes de la propriété. Vient alors la rubrique des équations (égalité entre termes, ex : *point(x, y) == point(y, x)*) qui permet de définir la sémantique des opérateurs. Les équations sont quantifiées universellement, c'est à dire qu'elles sont vraies pour toutes les valeurs des variables qui figurent dans les deux membres de l'égalité. On donne la sémantique des prédicats dans la rubrique des clauses (clauses de Horn avec égalité).

On trouve ensuite deux rubriques (*inherits* et *satisfies*) permettant d'établir des relations entre propriétés.

Enfin il est possible d'écrire des théorèmes dans la rubrique du même nom. Les théorèmes ont le même aspect que les équations (égalité entre deux termes) mais devront être démontrés à l'aide du démonstrateur pour pouvoir éventuellement être utilisés. Lors de leur déclaration, ils sont considérés, par défaut, comme n'étant pas démontrés.

Voyons maintenant la deuxième forme d'une propriété. Contrairement à la première, elle ne permet de construire une nouvelle propriété qu'à partir de propriétés déjà existantes.

```
property <module-name>
   sorts
      <formal-sort-name list>
   operators
      <formal-operator-declaration list>
   predicates
      <formal-predicate-declaration list>
   variables
      <variable-declaration list>
   combines
      <formal-model list>
   theorems
      <theorem list>
end <module-name>
```

La deuxième forme exprime le fait que la propriété ainsi définie est équivalente aux propriétés qu'elle combine. Le terme équivalent étant à manipulier avec précaution, précisons son sens dans notre contexte : une telle propriété va introduire une signature et le comportement des objets de la signature sera décrit à l'aide des équations et clauses provenant des propriétés combinées et grâce aux renommages précisés par la clause *combine*.

La différence entre cette forme et la précédente est, d'une part que les rubriques équations et clauses sont inexistantes (sinon l'équivalence ne serait pas forcément vérifiée) et d'autre part que les rubriques de déclarations de relations *inherits* et *satisfies* sont remplacées par une rubrique *combines* qui définit l'équivalence entre cette propriété et les propriétés combinées. Les autres rubriques ont toutefois la même signification.

Ces trois rubriques *satisfies*, *inherits* et *combines* permettent de définir des relations entre des propriétés LPG. Nous étudierons plus précisemment ces relations au cours du chapitre concernant les relations sémantiques.

Informellement, une clause *satisfies* P_s présente dans une propriété P_t indique que les équations de la propriété P_s doivent être des conséquences logiques (après renommage) de la théorie présentée par P_t.

Une clause *inherits* P_s écrite dans une propriété P_t signifie que les équations de la propriété P_s (après renommage) sont inclues dans la propriété P_t.

Une clause *combines* P_i, écrite dans une propriété P, a à peu près le même effet qu'une clause *inherits*. Elle permet de récupérer les équations d'autres

propriétés. Mais, à la différence d'une clause *inherits*, dès qu'un modèle de la propriété P est connu, le système déduit automatiquement les modèles des propriétés P_i. Réciproquement, si un modèle est connu pour chaque propriété P_i, le système va construire automatiquement un modèle de la propriété P.

Voyons sur un exmple l'utilisation d'une clause *combines*. Une première propriété (*Formal_Sort*) spécifie le comportement d'une sorte quelconque. On décrit ensuite (*Com*) la propriété de commutativité d'un opérateur, puis celle de l'associativité (*Assoc*). Enfin, on veut décrire le comportement d'un opérateur qui est à la fois commutatif et associatif.

```
property Formal_Sort                    — t will be a formal sort
sorts fs
end Formal_Sort

property Com
sorts s1
operators
  op1: (s1,s1)->s1
variables
  x, y: s1
equations
  1: op1(x,y) = op1(y,x)
satisfies
  Formal_Sort[s1]
end Com

property Assoc
sorts s2
operators
  op2: (s2,s2)->s2
variables
  x, y, z: s2
equations
  1: op2(x, op2(y, z)) = op2(op2(x, y), z)
satisfies
  Formal_Sort[s2]
end Assoc

property Assoc_Com
sorts s
operators
  op: (s,s) -> s
combines
  Com[s operators op],
  Assoc[s operators op]
end Assoc_Com
```

On crée pour cela une nouvelle propriété *Assoc_Com* dont la signature est $\Sigma_{Assoc_Com}=(\{s\}, \{op\}, \emptyset)$.

Les signatures de trois autres propriétés sont
- $\Sigma_{Formal_Sort}=(\{fs\}, \emptyset, \emptyset)$ pour la propriété *Formal_Sort*,
- $\Sigma_{Com}=(\{s1\}, \{op1\}, \emptyset)$ pour la propriété *Com* et
- $\Sigma_{Assoc}=(\{s2\}, \{op2\}, \emptyset)$ pour la propriété *Assoc* .

Les ensembles d'équations de ces trois propriétés sont
- $E_{Formal_Sort}=\emptyset$ pour la propriété *Formal_Sort*,
- $E_{Com}=\{op1(x,y) == op1(y,x)\}$ pour la propriété *Com* et
- $E_{Assoc}=\{op2(x,op2(y,z)) == op2(op2(x,y),z)\}$ pour la propriété *Assoc* .

La propriété *Assoc_Com* a alors l'ensemble d'équations suivant
- $E_{Assoc_Com}=\{op(x,y)==op(y,x),\ op(op(x,y),z)==op(x,op(y,z))\}$ où x, y, z sont trois variables de sorte *s*.

4.6- Les types

4.6.1- Les types non génériques

Un module LPG *type* est la définition algébrique d'un type abstrait. On introduit la définition de différentes sortes dont les valeurs sont obtenues, entre autre, par des opérateurs spéciaux appelés constructeurs (ex : *0* et *succ* sont les deux constructeurs des nombres naturels). Il est possible de définir d'autres opérateurs qui vont travailler sur ces valeurs.

De plus, un opérateur peut ne pas être défini sur la totalité de son domaine (la division n'est pas définie pour la valeur zéro, par exemple). Une construction de LPG, les exceptions, permet de prendre en compte ce type de problème. Lorsqu'on demande l'évaluation d'un opérateur et que l'un de ses arguments au moins est une valeur pour laquelle cet opérateur n'est pas défini et si, de plus, une exception a été spécifiée pour un tel cas, c'est cette dernière qui sera le résultat de l'évaluation de l'opérateur. Pratiquement, dans une telle situation, l'évaluation peut tout simplement s'arrêter en donnant un message d'erreur (l'évaluation de *1 / 0* émet le message *zero_divide:exception* qui traduit l'erreur). Ce message d'erreur est en fait le résultat du déclenchement de l'exception.

Généralement un opérateur est défini à l'aide d'autres opérateurs. Supposons que l'opérateur *g* soit défini à l'aide, entre autre, de l'opérateur *f* et que *f* ne soit pas défini sur tout son domaine, ce qui se traduira par le déclenchement d'une exception *e*. L'opérateur *g* peut prendre en compte dans sa spécification ce genre de problème. C'est ce qu'on appelle la récupération d'exception. Cette récupération peut s'effectuer de deux manières. Soit l'exception déclenchée par *f* est traitée par *g* (en présence de cette exception, l'opérateur *g* retourne le résultat de l'évaluation d'une expression particulière). Soit elle est propagée (l'opérateur *g* détecte le déclenchement de l'exception de la part de *f* et, à son tour, émet une autre exception qui sera éventuellement récupérée à un autre niveau).

Un type peut être générique. On dit qu'il exige une propriété qui décrit plus ou moins dans le détail le comportement d'objets formels (sortes, opérateurs ou prédicats) qui sont utilisés dans la spécification.

Un type non générique est décrit par une présentation $P_T=(\Sigma_T,E_T)$ où $\Sigma_T=(S_T,\Omega_T,\Pi_T)$ est la signature. La sémantique d'un tel module est l'algèbre $T_{\Sigma_T}/\equiv_{E_T}$ où $\equiv_{E_T}$ est la plus petite relation de congruence qui inclu E_T. Dans ce cas, la sémantique est dite initiale, elle correspond à la théorie inductive.

Présentons l'exemple de la spécification du type abstrait des valeurs booléennes.

```
type Bool
sorts bool
constructors
  true, false : -> bool          — The two possible values
operators                        — Specification of different operators
  not: bool-> bool
  and: (bool, bool)-> bool
  or:  (bool, bool)-> bool
  xor:  (bool, bool)-> bool
  =>: (bool, bool)-> bool
variables
  b,b1,b2: bool
equations
  1: not(true) ==> false
  2: not(false) ==> true
  3: true and b ==> b
  4: false and b ==> false
  5: b1 or b2 ==> not( not(b1) and not(b2) )
  6: b1 xor b2 ==> (b1 or b2) and not(b1 and b2)
  7: b1 => b2 ==> not(b1) or b2
end Bool
```

4.6.2- Les types génériques

4.6.2.1- Sémantique

Un type générique exige une propriété. Intuitivement, les définitions introduites par un tel type utiliseront des objets non connus à priori (formels), mais on supposera tout de même que ces objets possèdent des propriétés (au sens usuel du terme) particulières qui, elles, seront décrites dans des modules *propriété* de LPG. A l'utilisation, chaque définition présente dans un type générique sera ainsi paramétrée par les sortes, opérateurs et prédicats qui figurent dans la signature de la propriété exigée.

Un objet générique pourra être utilisé avec toutes définitions effectives de sortes, opérateurs et prédicats pourvu que ces dernières vérifient les comportements spécifiés par la propriété exigée.

Etant donnée une propriété P présentée par $P_P=(\Sigma_P,E_P)$ et un type LPG T présenté par $\dot{P}_T=(\dot{\Sigma}_T,\dot{E}_T)$[1] qui exige la propriété P, il existe un morphisme Φ^r entre P et T

$$P_P=(\Sigma_P,E_P) \xrightarrow{\Phi^r} P_T=(\Sigma_T,E_T).$$

Le morphisme Φ^r traduit le fait que la spécification du type T utilise des objets formels dont le comportement est spécifié dans la propriété P. Dans la théorie présentée par P_T, on trouve donc les axiomes issus de la présentation P_T, mais aussi les axiomes, après renommage, de la présentation de la propriété P.

En LPG, ce morphisme Φ^r test décrit de la manière suivante

$$P[S_1, S_2, ..., S_m \textbf{ operators } F_1, F_2, ..., F_n \textbf{ predicates } P_1, P_2, ..., P_q].$$

Dans cette notation, on retrouve le nom de la propriété P, la liste des sortes S_i, la liste des opérateurs F_j et la liste des prédicats P_k.

Le morphisme Φ^r est défini par $\Phi^r=(\Phi^r_S, \Phi^r_\Omega, \Phi^r_\Pi)$ où

- $\Phi^r_S : S_P \rightarrow S_T$,
- $\Phi^r_\Omega : \Omega_P \rightarrow \Omega_T$ et
- $\Phi^r_\Pi : \Pi_P \rightarrow \Pi_T$

sont trois fonctions. La première, Φ^r_S, fait correspondre les sortes de la propriété P (S_P) aux identificateurs de sortes ($S_{i,\ i=1,\ ...,\ m}$) de la déclaration du morphisme, qui font partie des sortes du type (S_T). De même, Φ^r_Ω est une fonction qui fait correspondre les opérateurs de la propriété P (Ω_P) aux opérateurs du types (Ω_T) désignés par la description du morphisme ($F_{j,\ j=1,\ ...,\ n}$). La troisième fonction, Φ^r_Π, de la même manière fait correspondre les prédicats de la propriété (Π_P) aux prédicats du types (Π_T) introduits par la description du morphisme ($P_{k,\ k=1,\ ...,\ q}$).

On voit alors que la théorie associée au type générique T est définie d'une part grâce aux axiomes de la présentation $\dot{P}_T$, mais également par les axiomes provenant de la présentation de la propriété P en utilisant le renommage introduit par le morphisme Φ^r.

---

[1]Présentation partielle.

Nous allons donc définir la sémantique d'un type LPG générique par le triplet (Th(P), Φ^r, Th(T)) où :

• Th(P) est la théorie équationnelle présentée par P_P,

• Φ^r est le morphisme défini dans la clause exige du type T. Ce morphisme doit être fortement persistent. Et

• Th(T) est la théorie inductive présentée par $(\Phi^r_S(S_P)\oplus\dot{S}_T)$, $\Phi^r_\Omega(\Omega_P)\oplus\dot{\Omega}_T$, $\Phi^r_\Pi(\Pi_P)\oplus\dot{\Pi}_T$, $\Phi^r_E(E_P)\oplus\dot{E}_T)$ avec les conditions $\Phi^r_S(S_P)\cap\dot{S}_T=\varnothing$, $\Phi^r_\Omega(\Omega_P)\cap\dot{\Omega}_T=\varnothing$ et $\Phi^r_\Pi(\Pi_P)\cap\dot{\Pi}_T=\varnothing$.

Le premier élément du triplet permet de connaître la théorie associée à la propriété exigée par le type.

Le troisième élément du triplet représente la théorie associée au type générique. On indique comment construire la présentation qui est à l'origine de cette théorie.

Enfin il est nécessaire de préciser le morphisme existant entre ces deux théories, ce qui est fait par le deuxième élément du triplet.

La présentation qui est à l'origine de la théorie Th(T) est composée des éléments suivants :

• $\Phi^r_S(S_P)\oplus\dot{S}_T$: on considère les sortes effectives déclarées dans le type T auxquelles on ajoute, après le renommage défini par la fonction Φ^r_S , les sortes provenant de la propriété P. LPG impose que les sortes renommées de la propriété doivent être différentes des sortes du type $(\Phi^r_S(S_P)\cap\dot{S}_T=\varnothing)$,

• $\Phi^r_\Omega(\Omega_P)\oplus\dot{\Omega}_T$: cet ensemble est constitué des opérateurs effectifs provenant du type T et des opérateurs renommés par la fonction Φ^r_Ω de la propriété P. LPG impose également que ces deux ensembles, $\Phi^r_\Omega(\Omega_P)$ et $\dot{\Omega}_T$, soient disjoints $(\Phi^r_\Omega(\Omega_P)\cap\dot{\Omega}_T=\varnothing)$,

• $\Phi^r_\Pi(\Pi_P)\oplus\dot{\Pi}_T$: on construit de la même manière l'ensemble des prédicats en faisant l'union entre l'ensemble des prédicats effectifs ($\dot{\Pi}_T$) du type générique T et l'ensemble des prédicats renommés par Φ^r_Π des prédicats de la propriété P. LPG impose aussi que ces deux ensembles, $\Phi^r_\Pi(\Pi_P)$ et $\dot{\Pi}_T$, soient

disjoints ($\Phi^r_\Pi(\Pi_P) \cap \dot{\Pi}_T = \varnothing$),

- $\Phi^r_E(E_P) \oplus \dot{E}_T$: enfin, l'ensemble des équations est construit en faisant

l'union entre l'ensemble des équations propres au type générique T ($\dot{E}_T$), c'est à dire qui figurent dans sa présentation, et l'ensemble des équations renommées par Φ^r_E des équations de la propriété P.

Dans notre cas, un type LPG est relié à une propriété LPG par cette clause *requires*. Nous définissons la sémantique d'un tel type générique par le triplet (Th(P), Φ^r, Th(T)).

Il existe, comme nous le verrons dans le prochain chapitre, d'autres relations entre les divers modules LPG. Dans chaque cas, nous définirons la sémantique de ces modules par l'intermédiaire d'un triplet (Th_1, Φ, Th_2).

La description de cette relation *requires* entre un module (type ou enrichissement) et une propriété est reprise dans le prochain chapitre et expliquée à l'aide d'un exemple complet.

Voyons sur un exemple très simple comment on décrit un type générique en LPG.

4.6.2.2- Description

L'exemple le type abstrait générique des séquences dont la sorte des éléments est inconnue à ce niveau mais dont le comportement est décrit dans la ptropriété *Formal_Sort*. On peut trouver d'autres exemples de modules génériques dans le chapitre décrivant les relations sémantiques.

```
property Formal_Sort
sorts t
end Formal_Sort

type Seq requires Formal_Sort[elem]
sorts seq
constructors
 nil: -> seq[elem]
 <+: (elem, seq[elem]) -> seq[elem]
end Seq
```

La propriété *Formal_Sort* décrit ce qu'est une sorte formelle. Les modèles de cette propriété sont des collections de valeurs. Le type des séquences *Seq* définit la manière de construire des séquences d'éléments dont on ne connaît pas la sorte à priori mais dont le comportement (très simple dans notre cas) est décrit par la propriété figurant dans la clause *requires*.

La forme générale d'un type en LPG est la suivante :

```
type <module-name>
  requires
    <formal-model>
  sorts
    <sort-name list>
  constructors
    <constructor-declaration list>
  operators
    <operator-declaration list>
  exceptions
    <exception-declaration list>
  predicates
    <predicate-declaration list>
  variables
    <variable-declaration list>
  equations
    <equation list>
  conditions
    <condition list>
  clauses
    <clause list>
  models
    <model list>
  theorems
    <theorem list>
end <module-name>
```

On a tout d'abord la rubrique introduisant la propriété exigée (***requires*** *Formal_Sort[elem* ***operators*** *leq, eq]*). Cette rubrique définit en fait un morphisme de signature entre la propriété exigée et le module courant. Il est à noter qu'il ne peut pas y avoir plus d'une propriété exigée. Cette restriction peut toutefois être contournée par l'utilisation des structurations des propriétés et, notamment le fait qu'une propriété puisse en combiner plusieurs autres.

La rubrique *sorts* définit les nouvelles sortes du type abstrait en cours de spécification.

La rubrique *constructors* est propre aux modules de type. On définit à cet endroit les différents opérateurs qui vont permettre de construire toutes les valeurs des sortes du type abstrait (ex : *true, false*).

La rubrique *operators* permet de spécifier la syntaxe d'opérateurs effectifs (alors que dans les propriétés, on caractérise une *classe* d'opérateurs).

On trouve ensuite la rubrique des *exceptions*. Elles permettent de signaler des cas d'erreur pouvant se produire pour des opérateurs partiellement définis sur leur domaine. Une exception peut être accompagnée de paramètres servant, par exemple, à communiquer les valeurs qui sont à l'origine de l'exception (ex : *index_fault : nat*).

Syntaxiquement, la rubrique des prédicats est la même que celle figurant dans une propriété. Sémantiquement on définit dans un type des prédicats effectifs alors que dans une propriété, on spécifie une *classe* de prédicats.

On retrouve une rubrique *variables* qui permet de déclarer les différentes variables qui seront utilisées dans les équations, conditions, clauses et théorèmes.

La rubrique *equations* est identique à celle des propriétés (égalités entre termes).

On trouve, par contre, une nouvelle rubrique *conditions* qui permet de spécifier les cas déclenchant les exceptions ainsi que les récupérateurs d'exception. Rappelons que les récupérateurs permettent, en présence d'une exception, soit de ne pas interrompre le calcul en cours substituant au résultat non défini de l'opérateur ayant causé l'exception le résultat de l'évaluation d'une expression, soit de propager l'exception à un autre niveau (éventuellement à la racine de l'expression générale).

La rubrique *clauses* donne la sémantique des prédicats précédemment déclarés en utilisant le formalisme des clauses de Horn avec égalité.

On trouve ensuite la rubrique permettant de définir de nouveaux modèles de propriétés. On a vu qu'une propriété permet de caractériser une classe d'objets. Opérationellement, on aura besoin d'un représentant particulier (instance) de ces objets. Un tel représentant est en fait la donnée des composantes effectives d'un objet (sorte, opérateurs et prédicats) pour chaque composante formelle. Une telle association est nommée modèle en LPG et définit en fait un morphisme entre la propriété et le module introduisant ce morphisme.

Enfin on retrouve la rubrique *theorems*, identique à celle présente dans les propriétés. Rappelons que pour être utilisables, ces théorèmes doivent être démontrés à l'aide de l'outil de démonstration.

4.7- Les enrichissements

4.7.1- Sémantique

Un module LPG *enrichissement* est assez semblable à un module *type*, en particulier, il peut être générique.

La différence réside dans le fait qu'il est interdit d'y déclarer des sortes et donc des constructeurs.

Par contre, du fait de l'importation (qui est implicite en LPG), il est possible de déclarer de nouveaux opérateurs sur des sortes déjà existantes (et donc déclarées par ailleurs dans des modules *types*) ou utilisant des opérateurs définis dans d'autres types ou enrichissements.

Un enrichissement permet donc de compléter des théories existantes par ajout de nouvelles fonctions.

Soit $P_U=(\Sigma_U, E_U)$ la présentation d'un module *type* ou *enrichissement* U. Soit $\dot{P}_{Enr}=(\dot{\Sigma}_{Enr},\dot{E}_{Enr})$ la présentation partielle d'un module *enrichissement* Enr qui

importe des déclarations définies dans U. Il existe un morphisme d'importation entre le module U et l'enrichissement Enr

$$P_U=(\Sigma_U,E_U) \xrightarrow{\Phi^i} P_{Enr}=(\Sigma_{Enr},E_{Enr}).$$

Ce morphisme exprime le fait que l'enrichissement Enr utilise les définitions présentes dans le module U. Dans la théorie présentée par P_{Enr}, on trouve les axiomes propres à Enr ainsi que les axiomes de la présentation du module U.

Il n'y a pas de notation spéciale en LPG pour déclarer les importations. Elle est implicite et le simple fait d'utiliser un objet non déclaré localement signifie l'importation de cet objet.

Le morphisme d'importation Φ^i est alors défini par $\Phi^i=(\Phi^i_S, \Phi^i_\Omega, \Phi^i_\Pi)$ où

- $\Phi^i_S : S_U \rightarrow S_{Enr}$,
- $\Phi^i_\Omega : \Omega_U \rightarrow \Omega_{Enr}$ et
- $\Phi^i_\Pi : \Pi_U \rightarrow \Pi_{Enr}$

sont trois fonctions qui incluent les sortes, opérateurs et prédicats du module importé U dans l'enrichissement Enr.

La théorie associée à l'enrichissement Enr contient donc les axiomes de la présentation $\dot{P}_{Enr}$, auxquels il faut ajouter les axiomes contenus dans la théorie présentée par P_U.

Nous définissons alors la sémantique d'un enrichissement LPG Enr comme étant le triplet (Th(U), Φ^i, Th(Enr)) où :

- Th(U) est la théorie inductive présentée par P_U,
- Φ^i est le morphisme identité persistent entre U et Enr et
- Th(Enr) est la théorie présentée par $(\Phi^i_S(S_U)\oplus\dot{S}_{Enr}, \Phi^i_\Omega(\Omega_U)\oplus\dot{\Omega}_{Enr}, \Phi^i_\Pi(\Pi_U)\oplus\dot{\Pi}_{Enr}, \Phi^i_E(E_U)\oplus\dot{E}_{Enr})$ avec les conditions $\Phi^i_S(S_U)\cap\dot{S}_{Enr}=\varnothing$, $\Phi^i_\Omega(\Omega_U)\cap\dot{\Omega}_{Enr}=\varnothing$ et $\Phi^i_\Pi(\Pi_U)\cap\dot{\Pi}_{Enr}=\varnothing$.

On considère la théorie associée au module importé (premier élément du triplet), le morphimes identité (deuxième élément) puis la théorie associée à l'enrichissement (troisième élément du triplet).

Puisque le morphisme Φ^i est le morphisme identité, la présentation

$$(\Phi^i_S(S_U)\oplus\dot{S}_{Enr}, \Phi^i_\Omega(\Omega_U)\oplus\dot{\Omega}_{Enr}, \Phi^i_\Pi(\Pi_U)\oplus\dot{\Pi}_{Enr}, \Phi^i_E(E_U)\oplus\dot{E}_{Enr})$$

peut s'écrire plus simplement

$$(S_U \oplus \dot{S}_{Enr},\ \Omega_U \oplus \dot{\Omega}_{Enr},\ \Pi_U \oplus \dot{\Pi}_{Enr},\ E_U \oplus \dot{E}_{Enr})$$

Ce qui revient à dire que l'on fait l'union entre les sortes (respectivement opérateurs, prédicats et équations) du module importé et les sortes (respectivement opérateurs, prédicats et équations) propres à l'enrichissement.

Les conditions

- $\Phi^i_S(S_U) \cap \dot{S}_{Enr} = \emptyset$ (1)
- $\Phi^i_\Omega(\Omega_U) \cap \dot{\Omega}_{Enr} = \emptyset$ (2)
- $\Phi^i_\Pi(\Pi_U) \cap \dot{\Pi}_{Enr} = \emptyset$ (3)

indiquent que les sortes (respectivement opérateurs et prédicats) du module importé doivent être différents des sortes (respectivement opérateurs et prédicats) propres à l'enrichissement.

Un enrichissement n'ayant pas de sorte propre ($\dot{S}_{Enr} = \emptyset$), la condition (1) est vérifiée par définition.

Les conditions (2) et (3) sont quant à elles vérifiées par construction du langage. En effet, il n'y aura importation que si un objet utilisé dans un enrichissement n'est pas défini dans cet enrichissement. De plus on ne peut pas définir en LPG deux objets différents ayant le même identificateur[1].

La relation d'importation est également décrite dans le chapitre concernant les relations sémantiques et illustrée à l'aide d'un exemple complet.

4.7.2- Description

L'exemple suivant définit deux opérateurs (= et /=) sur la sorte *bool* qui est spécifiée par ailleurs dans un type. N'étant pas déclarés dans cet enrichissement, la sorte *bool* et les opérateurs *true*, *false* et *not* sont importés.

```
enrichment Eq_Bool
operators
  =, /=: (bool, bool) -> bool
variables
  b1, b2: bool
equations
  1: true = true ==> true
  2: true = false ==> false
```

[1]La surcharge des opérateurs et des prédicats est toutefois autorisée. On peut considérer qu'un opérateur (respectivement un prédicat) n'est pas indentifié que par son identificateur, mais également par son domaine et son codomaine (respectivement domaine).

```
    3: false = true ==> false
    4: false = false ==> true
    5: b1 /= b2 ==> not(b1 = b2)
end Eq_Bool
```

Le squelette d'un enrichissement est inclus dans celui d'un type. On retrouve donc les mêmes rubriques exceptées *sortes* et *constructors*.

```
enrichment <module-name>
   requires
      <formal-model>
   operators
      <operator-declaration list>
   exceptions
      <exception-declaration list>
   predicates
      <predicate-declaration list>
   variables
      <variable-declaration list>
   equations
      <equation list>
   conditions
      <condition list>
   clauses
      <clause list>
   models
      <model list>
   theorems
      <theorem list>
end <module-name>
```

4.8- Bibliothèque prédéfinie

Dans l'atelier de spécification LPG, il est possible de définir, ou plus généralement d'utiliser des spécifications et des programmes. On peut donc considérer que cet atelier contient à tout instant un certain nombre de définitions disponibles pour tout utilisateur.

On appelle *bibliothèque* cet ensemble de définitions. Lorsqu'on entre pour la première fois dans l'atelier LPG, il existe un ensemble minimal de définitions appelés *bibliothèque prédéfinie*.

On peut trouver en annexe, la signature des différentes spécifications contenues dans la bibliothèque prédéfinie. Nous allons simplement énumérer, ici, les propriétés et sortes spécifiées.

En ce qui concerne les propriétés, nous trouvons celle caractérisant une sorte formelle. On trouve aussi la commutativité et l'associativité d'un opérateur. Une structure de monoïde est définie. Les relations d'égalités, d'ordre partiel et d'ordre total sont également spécifiées.

Les sortes suivantes se trouvent également dans la bibliothèque prédéfinie. Les booléens, les entiers naturels, les caractères ainsi que les chaînes de caractères, les séquences, les tableaux et les ensembles d'éléments.

On peut enfin y trouver différentes fonctions relatives aux entrées-sorties.

4.9- Liste des commandes

quitter : permet de quitter l'atelier de spécification.

compiler <file_name> <options> : permet de compiler un fichier contenant des définitions LPG. Parmi les différentes options (une option est un mot-clé précédé par le caractère tiret), on peut citer celle *-list* qui donne une liste de la compilation (listing) et *-temps* qui donne le temps d'exécution de la commande de compilation.

evaluer <file_name> <options> : offre la possibilité d'évaluer des expressions symboliques (réduction sous forme normale), des expressions logiques (si une expression logique ne contient pas de variable, le résultat indique si l'expression est une conséquence des spécifications, sinon le résultat est la liste des valeurs des variables pour lesquelles l'expression logique est une conséquence des spécifications) et de faire des déclarations de variables (utilisées pour l'évaluation d'expression logiques). Les principales options sont les mêmes que celles de la commande *compiler*.

visualiser <options> : donne différents renseignements sur l'état de la bibliothèque. On peut ainsi obtenir le taux de remplissage des tables internes, voir l'intégralité de la bibliothèque ou examiner globalement ou dans le détail certaines parties des spécifications (sortes, opérateurs, etc...).

sauvegarder <file_name> : crée une bibliothèque qui contiendra les définitions courantes de l'atelier.

restaurer : remet les tables internes de LPG dans l'état dans lequel elles se trouvaient au tout début de la session.

dump <file_name> : permet de visualiser directement l'état interne des tables de LPG.

valider : lance une session de vérification sémantique de modules LPG.

prouver : démarre une session de démonstration de théorèmes figurant dans les rubriques de même nom des différents modules.

5- Relations sémantiques

5.1- Introduction

A l'origine de ce travail, nous avons voulu exhiber certaines conditions qui doivent être vérifiées pour que l'on puisse dire d'une spécification qu'elle est sémantiquement valide. Comme nous l'avons déjà vu, ces conditions sont représentées par des formules logiques qui doivent être des théorèmes de la théorie associée à la spécification.

Plus précisemment, nous avons considéré le fait qu'une spécification peut s'écrire en plusieurs modules, chaque module pouvant être relié à d'autres par une relation implicite ou explicite. On est alors ammené à voir la théories associée à une spécification, écrite sous forme de plusieurs modules, comme une théorie structurée à l'aide de différents morphismes. Cette étude s'est basée sur le langage LPG, présenté dans le chapitre précédent.

Les spécifications LPG sont, en effet, écrites hiérarchiquement en utilisant la notion de modules reliés éventuellement les uns aux autres par certaines relations. On peut voir ces relations comme des morphismes de théorie.

Un morphisme $\Phi=(\Phi_S, \Phi_P, \Phi_\Pi)$ d'un module *P* présenté par $P_P=(\Sigma_P,E_P)$ vers un module *U* présenté par $P_U=(\Sigma_U,E_U)$, peut être représenté de la manière suivante

$$P_P=(\Sigma_P,E_P) \xrightarrow{\Phi} P_U=(\Sigma_U,E_U).$$

Il fait correspondre des sortes, opérateurs et prédicats du module *P* à des sortes, opérateurs et prédicats du module *U* par l'intermédiaire des trois fonctions $\Phi_S : S_P \rightarrow S_U$, $\Phi_\Omega : \Omega_P \rightarrow \Omega_U$ et $\Phi_\Pi : \Pi_P \rightarrow \Pi_U$.

Le module *U* sera parfois appelé le module cible du morphisme. De même, on nommera parfois le module *P* module source du morphisme.

En LPG, un tel morphisme sera généralement décrit dans le module *U* par

$$P[S_1, S_2, ..., S_m \textbf{ operators } F_1, F_2, ..., F_n \textbf{ predicates } P_1, P_2, ..., P_q].$$

On trouve tout d'abord le nom du module *P* qui est à la source du morphisme. Viennent ensuite les sortes (S_i) qui sont les images des sortes du module *P* par la partie du morphisme relative aux sortes Φ_S. Le mot-clé *operators* introduit de la même manière les images (F_j) des opérateurs par Φ_P. Enfin, le mot-clé *predicates* introduit les images (P_k) des prédicats par Φ_Π.

Il existe en LPG sept types de relations divisés en deux groupes, celles entre propriétés et les autres. Ce sont les relations de *satisfaction*, d'*héritage* et de

combinaison en ce qui concerne les propriétés et l'*importation*, la *paramétrisation*, la *déclaration de modèle* et l'*instanciation* pour les autres.

On peut remarquer que l'ensemble des modules avec les différentes relations qui les relient forme un graphe acyclique avec les modules comme nœuds et les relations comme arcs. On peut donc écrire U_s—rel→U_t et parler de U_s comme se trouvant en amont de U_t (ou réciproquement, de U_t qui se trouve en aval de U_s).

Nous avons vu qu'à une signature peuvent correspondre plusieurs signatures partielles. Nous allons appeler *la signature partielle*, la signature ne contenant que les sortes/opérateurs/prédicats introduits par une unité LPG.

Nous allons maintenant formaliser ces différentes relations. Chacune de ces formalisations suppose l'existence d'une seule relation entre les unités. S'il existe plusieurs types de relations, la construction de la présentation de la théorie cible doit se faire incrémentalement en ne considérant plus la signature partielle associée à la théorie cible, mais une signature partielle "plus complète".

Chaque relation y est décrite formellement en donnant la sémantique du module cible de la relation. Nous utilisons la même notion de sémantique que celle vue dans le chapitre précédent concernant la sémantique des types génériques, par exemple.

La théorie associée au module *U* est définie d'une part grâce aux axiomes de la présentation $\dot{P}_U$, mais également par les axiomes provenant de la présentation du module *P* en utilisant le renommage introduit par le morphisme Φ.

La sémantique d'un module LPG qui est la cible de l'une des sept relations est donc donnée par le triplet (Th(P), Φ, Th(U)) où :

- Th(P) est la théorie équationnelle présentée par P_P,
- Φ est le morphisme associé à la relation et
- Th(U) est la théorie associée au module cible de la relation.

Le premier élément du triplet nous donne la théorie associée au module source de la relation.

Le deuxième élément est le morphisme lui-même.

Le troisième élément du triplet représente la théorie associée au module cible de la relation. On indique généralement à ce niveau comment construire la présentation qui est à l'origine de cette théorie.

De plus, les conditions à vérifier pour s'assurer de la validité sémantique des spécifications seront trouvées généralement dans cette troisième partie du triplet.

Chacune des descriptions formelles de ces relations est accompagnée d'un ou plusieurs exemples détaillés.

Enfin, nous avons présenté dans le chapitre précédent les relations d'importation et de paramétrisation dans les cas particuliers des descriptions respectivement d'enrichissement et de types génériques. Ces deux genres de relations sont généralisées dans ce chapitre.

5.2- Importation

L'importation est le fait de pouvoir utiliser dans un module des objets (sortes, opérateurs ou prédicats) qui ont été définis dans des modules de type ou d'enrichissement.

Cette notion d'importation a déjà été rencontrée lors de la description des modules d'enrichissement. La description formelle donnée dans le cas d'un enrichissement est un cas particulier de la description que nous donnons dans cette partie. L'importation de définitions est implicite en LPG. Toutes déclarations est donc visible de l'extérieur[1].

Dans l'exemple ci-dessous, nous définissons les types abstraits des booléens et des naturels.

```
type Bool
sorts bool
constructors
  true : -> bool
  false : -> bool
operators
  not: bool-> bool
  and: (bool, bool)-> bool
  or:  (bool, bool)-> bool
variables
  b, b1, b2: bool
equations
  1: not(true) ==> false
  2: not(false) ==> true
  3: true and b ==> b
  4: false and b ==> false
  5: b1 or b2 ==> not( not(b1) and not(b2) )
end Bool
```

La signature de ce module est

$$\Sigma_{Bool}=(<\emptyset,\{bool\}>, <\emptyset,\{true, false, not, and, or\}>, <\emptyset,\emptyset>).$$

Par ailleurs, on peut constater que tout ce qui est utilisé est défini dans le même module. Il n'importe donc rien.

```
type Natural
sorts nat
constructors
  0 : -> nat
  succ: nat -> nat
```

[1]Il est toutefois possible de masquer une définition d'opérateur si l'utilisateur le juge nécessaire.

```
operators
  +, *: (nat, nat) -> nat
  = : (nat, nat) -> bool
variables
  n, m, p: nat
equations
  1: 0+n ==> n
  2: succ(m) + n ==> succ(m+n)
  3: 0*n ==> 0
  4: succ(m) *n ==> n+(m*n)
  5: 0 = 0 ==> true
  6: 0 = succ(m) ==> false
  7: succ(n) = 0 ==> false
  8: succ(n) = succ(m) ==> n = m
end Natural
```

La signature partielle du module des naturels est

$$\dot{\Sigma}_{Natural}=(<\emptyset, \{nat\}>, <\emptyset, \{0, succ, +, *, =\}>, <\emptyset, \emptyset>).$$

Sa signature est

$$\Sigma_{Natural}=(<\emptyset, \{nat, bool\}>, <\emptyset, \{0, succ, +, *, =, true, false, not, and, or\}>, <\emptyset, \emptyset>).$$

Dans les déclarations, on voit que l'opérateur = prend deux arguments de sorte *nat* et donne un résultat de sorte *bool*. Or la sorte *bool* a été définie dans un autre module *Bool*. On peut donc dire que le module *Natural* importe les définitions du module *Bool*.

Soit U_s un module LPG présenté par $P_{U_s}=(\Sigma_{U_s},E_{U_s})$ et U_t un autre module LPG présenté par $\dot{P}_{U_t}=(\dot{\Sigma}_{U_t},\dot{E}_{U_t})$. La relation d'importation est alors notée $U_s \xrightarrow{i} U_t$ et la sémantique du module cible, U_t, d'une telle relation d'importation est donnée par le triplet $(Th(U_s), \Phi^i, Th(U_t))$ où :

- $Th(U_s)$ est la théorie présentée par P_{U_s},
- Φ^i est le morphisme identité entre $Th(U_s)$ et $Th(U_t)$ et
- $Th(U_t)$ est la théorie présentée par $(\Phi^i_S(S_{U_s}) \oplus \dot{S}_{U_t}, \Phi^i_\Omega(\Omega_{U_s}) \oplus \dot{\Omega}_{U_t}, \Phi^i_\Pi(\Pi_{U_s}) \oplus \dot{\Pi}_{U_t}, \Phi^i_E(E_{U_s}) \oplus \dot{E}_{U_t})$ avec les conditions

- $\Phi^i_S(S_{U_s}) \cap \dot{S}_{U_t}=\emptyset,$ (1)

- $\Phi^{i}_{\Omega}(\Omega_{U_s}) \cap \dot{\Omega}_{U_t} = \emptyset$ et (2)
- $\Phi^{i}_{\Pi}(\Pi_{U_s}) \cap \dot{\Pi}_{U_t} = \emptyset$. (3)

Le morphisme Φ étant le morphisme identité, la présentation de la théorie $Th(U_t)$ peut s'écrire plus simplement

$$(S_{U_s} \oplus \dot{S}_{U_t}, \Omega_{U_s} \oplus \dot{\Omega}_{U_t}, \Pi_{U_s} \oplus \dot{\Pi}_{U_t}, E_{U_s} \oplus \dot{E}_{U_t}).$$

Les trois conditions expriment le fait que les sortes, opérateurs et prédicats du module importé U_s ne doivent pas exister déjà dans le module importateur U_t. Ces conditions sont vérifiées par construction du langage. En effet, l'importation est implicite en LPG et ne se produit que si un objet utilisé dans un module n'est pas déclaré à l'intérieur de ce module. De plus, deux objets différents ne peuvent pas avoir le même identificateur[1].

Il faut faire attention dans le cas de l'importation d'une unité paramétrée. La sémantique de LPG impose certaines contraintes concernant les modèles manipulés.

Il est nécessaire, d'une part que l'unité importatrice soit elle-même paramétrée et, d'autre part que le morphisme définissant la paramétrisation de l'unité importée soit inclus, à un renommage près dans le morphisme définissant la paramétrisation de l'unité importatrice.

Informellement, on peut dire que l'importation d'une unité paramétrée se traduit surtout par l'importation des déclarations non formelles. Ce cas est étudié plus en détail au cours de la description de la relation de paramétrisation.

En reprenant l'exemple, le morphisme est le suivant :

- Φ^{i}_{S} : bool→bool,
- Φ^{i}_{Ω} : true→true, false→false, not→not, and→and, or→or et
- Φ^{i}_{Π} : $\emptyset \rightarrow \emptyset$.

La signature de la présentation de la théorie cible est alors :

- $S_{Bool} \oplus \dot{S}_{Natural}$={bool, nat},
- $\Omega_{Bool} \oplus \dot{\Omega}_{Natural}$={true, false, not, and, or, 0, succ, +, *, =},
- $\Pi_{Bool} \oplus \dot{\Pi}_{Natural} = \emptyset$.

[1]En ce qui concerne la surcharge des opérateurs (respectivement des prédicats), on peut considérer qu'un tel objet est identifié par son identificateur et par son domaine et codomaine (respectivement domaine).

5.3- Paramétrisation

5.3.1- Cas général

La paramétrisation d'un module U (*type* ou *enrichment*) par une propriété P consiste à exprimer le fait que l'utilisation des objets définis dans U est fonction d'autres objets non connus à priori dans ce module U, mais dont le comportement général peut être décrit dans la propriété P.

Regardons, par exemple, la définition d'un opérateur d'égalité = entre des séquences d'éléments.

```
enrichment Seq_Equality
requires Equality [elem operators eq]
operators
   = : (seq[elem],seq[elem]) -> bool
variables
  e1, e2: elem
  s1, s2: seq[elem]
equations
  1: nil:seq[elem] = nil:seq[elem] ==> true
  2: e1 <+ s1 = nil:seq[elem] ==> false
  3: nil:seq[elem] = e2 <+ s2 ==> false
  4: e1 <+ s1 = e2 <+ s2 ==> if eq(e1, e2) then
                               s1 = s2
                             else
                               false
                             endif
end Seq_Equality
```

La signature partielle de ce module est

$$\dot{\Sigma}_{Seq_Equality}=(<\emptyset, \emptyset>, <\emptyset, \{=\}>, <\emptyset, \emptyset>).$$

On voit tout d'abord que cet opérateur d'égalité prend deux séquences d'éléments comme arguments et retourne un résultat booléen.

Sémantiquement, il est défini par cas. Deux séquences vides sont égales (équation 1). Une séquence vide est différente de tout séquence non vide (équations 2 et 3). Enfin, deux séquences non vides seront égales si le premier élément de la première séquence est égal au premier élément de la deuxième séquence et que le reste de chacune des séquences est identique.

Sur cet exemple, on voit que l'opérateur d'égalité entre des séquences est défini à partir d'un opérateur d'égalité entre les éléments de la séquence (équation 4) qui n'est pas connu dans cette définition mais dont le comportement général est décrit dans la propriété exigée *Equality* dont voici la spécification.

```
property Equality
sorts t
```

```
operators
   =: (t,t)->bool
variables
   x,y,z: t
equations
  1: x=x == true
  2: x=y == y=x
  3: (x=y and y=z) => x=z == true
satisfies
  Formal_Sort[t]
end Equality
```

La signature partielle de cette propriété est

$$\dot{\Sigma}_{Equality}=(<\emptyset, \{t\}>, <\emptyset, \{=\}>, <\emptyset, \emptyset>).$$

Sa signature, en tenant compte de l'importation des booléens est

$$\Sigma_{Equality}=(<\emptyset, \{bool, t\}>, <\emptyset, \{true, false, not, and, or, =\}>, <\emptyset, \emptyset>).$$

Cette propriété exprime simplement que le symbole noté = opérant sur des éléments d'une sorte quelconque *t* est réflexif, symétrique et transitif (équations 1, 2 et 3).

Voici maintenant la spécification des séquences

```
type Seq
requires Formal_Sort[s]
sorts seq
constructors
   nil : -> seq[s]
   <+ : (t, seq[s]) -> seq[s]
end Seq
```

Sa signature partielle est

$$\dot{\Sigma}_{Seq}=(<\emptyset, \{seq\}>, <\emptyset, \{nil, <+\}>, <\emptyset, \emptyset>).$$

La relation introduite entre un module propriété et un autre module générique par une clause *requires* est appelée relation de paramétrisation. Cette relation a déjà été vue lors de la description des types LPG génériques. Cette relation est ici généralisée pour un enrichissement ou un type générique.

Soit U un module LPG présenté par $\dot{P}_U=(\dot{\Sigma}_U,\dot{E}_U)$ qui exige une propriété P présentée par $P_P=(\Sigma_P,E_P)$. La relation de paramétrisation s'écrit alors $P \xrightarrow{r} U$ et elle introduit un morphisme Φ^r

$$P_P=(\Sigma_P,E_P) \xrightarrow{\Phi^r} P_U=(\Sigma_U,E_U).$$

Cette relation, en LPG, se déclare dans le module *U* et s'écrit

***requires** $P[S_1, S_2, ..., S_m$ **operators** $F_1, F_2, ..., F_n$ **predicates** $P_1, P_2, ..., P_q]$.*

On indique derrière le mot-clé *requires* le nom de la propriété exigée *P*, puis les images par Φ^r_S (respectivement Φ^r_Ω et Φ^r_Π) des sortes (respectivement opérateurs et prédicats) de cette propriété *P*.

La sémantique d'un module cible d'une relation de paramétrisation est donnée par le triplet (Th(P), Φ^r, Th(U)) où :

- Th(P) est la théorie présentée par P_P,
- Φ^r est le morphisme fortement persistent défini par la notation de la clause *requires* et
- Th(U) est la théorie présentée par $(\Phi^r_S(S_P)\oplus\dot{S}_U, \Phi^r_\Omega(\Omega_P)\oplus\dot{\Omega}_U, \Phi^r_\Pi(\Pi_P)\oplus\dot{\Pi}_U, \Phi^r_E(E_P)\oplus\dot{E}_U)$ sous les conditions

$$\bullet\ \Phi^r_S(S_P)\cap\dot{S}_U=\emptyset, \tag{1}$$

$$\bullet\ \Phi^r_\Omega(\Omega_P)\cap\dot{\Omega}_U=\emptyset \text{ et} \tag{2}$$

$$\bullet\ \Phi^r_\Pi(\Pi_P)\cap\dot{\Pi}_U=\emptyset. \tag{3}$$

Les trois conditions expriment le fait que les sortes, opérateurs et prédicats de la propriété *P*, après le renommage spécifié par *Φ*, ne doivent pas exister déjà dans le module *U*. Ces conditions sont vérifiées par construction du langage.

En considérant l'exemple de la spécification des séquences et sachant que la signature de la propriété *Formal_Sort* est

$$\Sigma_{Formal_Sort}=(<\emptyset, \{t\}>, <\emptyset, \emptyset>, <\emptyset, \emptyset>),$$

la signature du type *Seq* est

$$\Sigma_{Seq}=(<\{s\}, \{s, seq\}>, <\emptyset, \{nil, <+\}>, <\emptyset, \emptyset>).$$

Le morphisme de signature utilisé entre la propriété *Formal_Sort* et le type *Seq* est le suivant :

- $t\rightarrow s$ pour les sortes,
- $\emptyset\rightarrow\emptyset$ pour les opérateurs et
- $\emptyset\rightarrow\emptyset$ pour les prédicats.

En ce qui concerne l'égalité des séquences. Le morphisme de signature $\Phi^r_\Sigma=(\Phi^r_S, \Phi^r_\Omega, \Phi^r_\Pi)$ entre la propriété *Equality* et l'enrichissement *Seq_Equality* établit la correspondance :

- Φ^r_S : t→elem, bool→bool pour les sortes,
- Φ^r_Ω : =→eq, true→true, false→false, not→not, and→and, or→or

pour les opérateurs et

- Φ^r_Π : Ø→Ø pour les prédicats.

On obtient ainsi une signature partielle pour la présentation de la théorie Th(U) :

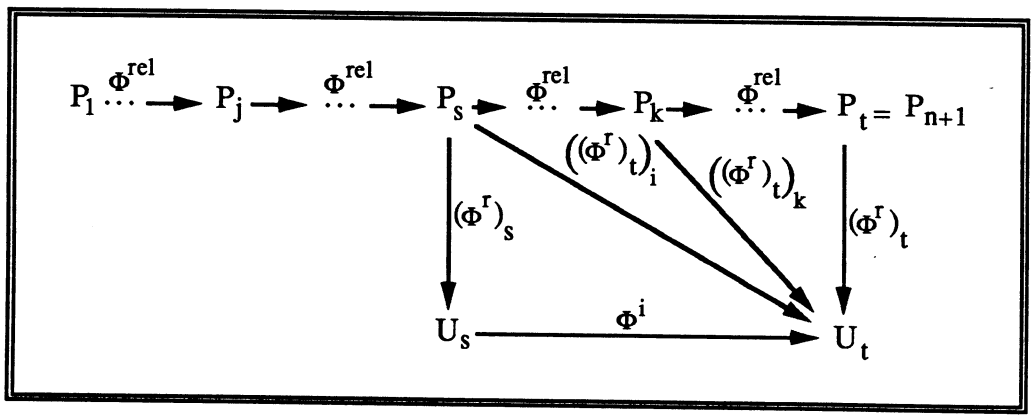
- $\Phi^r_S(S_{Equality})\oplus\dot{S}_{Seq_Equality}=\{elem, bool\}$,
- $\Phi^r_\Omega(\Omega_{Equality})\oplus\dot{\Omega}_{Seq_Equality}=\{eq, =, true, false, not, and, or\}$ et
- $\Phi^r_\Pi(\Pi_{Equality})\oplus\dot{\Pi}_{Seq_Equality}=\emptyset$.

Il faut remarquer que cette signature est à nouveau partielle. En effet, les séquences n'y apparaissent pas.

5.3.2- Importation d'un module paramétré

Voyons ce qui se passe maintenant dans le cas de l'importation d'une unitée paramétrée.

Soit U_s un module LPG présenté par $P_{U_s}=(\Sigma_{U_s},E_{U_s})$ et U_t un autre module LPG présenté par $\dot{P}_{U_t}=(\dot{\Sigma}_{U_t},\dot{E}_{U_t})$. La relation d'importation est notée $U_s \xrightarrow{i} U_t$. Si le module U_s est paramétré par une propriété P_s, il est nécessaire suivant la sémantique de LPG que le module U_t soit paramétré par une propriété P_t telle que P_s soit la même propriété que P_t ou bien que P_s soit en amont de P_t dans le graphe reliant les différentes propriétés entre elles.



Importation d'un module paramétré

La paramétrisation de U_t peut se noter $P_t \xrightarrow{r} U_t$ et donne naissance à un morphisme $(\Phi^r)_t$. Celle de U_s, $P_s \xrightarrow{r} U_s$ génère un morphisme $(\Phi^r)_s$.

La déclaration de $(\Phi^r)_t$ engendre la définition de n ($n \geq 0$) morphismes entre chacune des n propriétés se trouvant en amont de P_t et le module U_t.

Si n=0, les deux propriétés exigées sont identiques. La sémantique d'un module U_t qui est la cible d'une telle relation d'importation est alors donnée par le triplet $(Th(U_s), \Phi^i, Th(U_t))$ où :

- $Th(U_s)$ est la théorie présentée par P_{U_s},
- Φ^i est le morphisme identité entre $Th(U_s)$ et $Th(U_t)$ et
- $Th(U_t)$ est la théorie présentée par $((S_{U_s} \Theta S^f_{U_s}) \oplus \dot{S}_{U_t}, (\Omega_{U_s} \Theta \Omega^f_{U_s}) \oplus \dot{\Omega}_{U_t}, (\Pi_{U_s} \Theta \Pi^f_{U_s}) \oplus \dot{\Pi}_{U_t}, (\Phi^r_E)_t\left((\Phi^r_E)^{-1}_s(E_{U_s})\right) \oplus \dot{E}_{U_t})$.

$S_{U_s} \Theta S^f_{U_s}$ représente les sortes non formelles de U_s, $\Omega_{U_s} \Theta \Omega^f_{U_s}$ représente les opérateurs non formels de U_s et $\Pi_{U_s} \Theta \Pi^f_{U_s}$ représente les prédicats non formels de U_s.

$(\Phi^r_E)_t\left((\Phi^r_E)^{-1}_s(E_{U_s})\right)$ permet de traduire les équations présentes dans U_s en des équations dans U_t.

En effet, que le module importé soit paramétré ou non, l'importation des objets non formels ne pose pas de problème. C'est pour cela que la signature Σ_{U_t} est constituée de la signature partielle $\dot{S}_{U_t}$ complétée d'une part par la signature provenant de la propriété exigée $(\Phi^r_\Sigma)_t(\Sigma_{P_t})$ et d'autre part par les éléments non formels de la signature du module importé $\Sigma_{U_s} \Theta \Sigma^f_{U_s}$.

En ce qui concerne les équations, la situation est un peu plus compliquée puisqu'elles manipulent aussi bien les objets formels que les objets non formels. Du fait de la sémantique du langage, les objets formels manipulés dans le module U_s sont identiques aux objets formels manipulés dans le module U_t. L'inclusion des équations de U_s doit donc se réaliser de la manière suivante :

- renommage des objets formels selon $(\Phi^r_E)^{-1}_s$, on obtient ainsi les images figurant dans la propriété P des objets formels du module U_s.
- renommage de ces objets selon $(\Phi^r_E)_t$ pour les inclure effectivement dans le module U_t.

Si $n \geq 1$ alors il existe n morphismes $\left((\Phi^r)_t\right)_{i,i=1,\ldots,n}$ entre les n propriétés $P_{i,i=1,\ldots,n}$ se situant en amont de P_t.

La sémantique de LPG impose qu'il existe k, $1 \leq k \leq n$, tel que P_s soit la même propriété que P_k. Soit $\left((\Phi^r)_t\right)_k$ le morphisme correspondant entre la propriété $P_s = P_k$ et le module U_t, la sémantique du module cible U_t de cette relation d'importation est alors donnée par le triplet $(Th(U_s), \Phi^i, Th(U_t))$ où :

- $Th(U_s)$ est la théorie présentée par P_{U_s},
- Φ^i est le morphisme identité entre $Th(U_s)$ et $Th(U_t)$ et
- $Th(U_t)$ est la théorie présentée par $((S_{U_s} \Theta S^f_{U_s}) \oplus \dot{S}_{U_t}, (\Omega_{U_s} \Theta \Omega^f_{U_s}) \oplus \dot{\Omega}_{U_t}, (\Pi_{U_s} \Theta \Pi^f_{U_s}) \oplus \dot{\Pi}_{U_t}, \left((\Phi^r_E)_t\right)_k \left((\Phi^r_E)^{-1}_s (E_{U_s})\right) \oplus \dot{E}_{U_t})$.

$S_{U_s} \Theta S^f_{U_s}$, $\Omega_{U_s} \Theta \Omega^f_{U_s}$ et $\Pi_{U_s} \Theta \Pi^f_{U_s}$ ont la même signification que précédemment.

$\left((\Phi^r_E)_t\right)_k \left((\Phi^r_E)^{-1}_s (E_{U_s})\right)$ permet, de la même manière que dans le cas où n=0, de traduire les équations présentes dans U_s en des équations dans U_t.

Dans l'exemple de l'égalité entre les séquences, le module *Seq_Equality* est paramétré par la propriété *Equality* et importe la spécification *Seq* qui est paramétrée par la propriété *Formal_Sort*. Cette dernière propriété se trouve bien en amont de *Equality* (cf. clause *satisfies*). Le morphisme qui traduirait[1] les équations de *Formal_Sort* serait défini de la manière suivante.

On renomme tout d'abord les objets formels de Seq selon $(\Phi^r)^{-1}_s$:

- $(\Phi^r_S)^{-1}_s (S_{Seq})$: s→t, seq→seq,
- $(\Phi^r_\Omega)^{-1}_s (\Omega_{Seq})$: nil→nil, <+→<+ et
- $(\Phi^r_\Pi)^{-1}_s (\Pi_{Seq})$: Ø→Ø.

On renomme alors le résultat précédent selon $\left((\Phi^r)_t\right)_k$:

- $\left((\Phi^r_S)_t\right)_k \left((\Phi^r_S)^{-1}_s (S_{Seq})\right)$: s→elem, seq→seq,

[1]Le module *Formal_Sort* ne contient aucune équation.

- $\left((\Phi^r_\Omega)_t\right)_k\left((\Phi^r_\Omega)^{-1}_s(\Omega_{Seq})\right)$: nil→nil, <+→<+ et
- $\left((\Phi^r_\Pi)_t\right)_k\left((\Phi^r_\Pi)^{-1}_s(\Pi_{Seq})\right)$: Ø→Ø.

En tenant compte ainsi de l'importation des séquences, on a la signature de *Seq_Equality* :

$$\Sigma_{Seq_Equality}=(<\{elem\}, \{elem, seq, bool\}>, <\{eq\}, \{eq, nil, <+, =, true, false, not, and, or\}>, <\emptyset, \emptyset>).$$

5.4- Déclaration de modèles

Les modules propriété spécifient des classes d'objets (sortes, opérateurs et prédicats). En pratique, ce sont des instances particulières (des exemplaires des objets déclarés dans une propriété) qui seront utilisées.

Une déclaration de modèle permet de créer ces instances particulières.

L'exemple qui suit spécifie des opérateurs d'égalité et de différence sur les valeurs booléennes. On déclare, ensuite, un modèle de l'égalité sur les booléens et un autre sur les entiers naturels.

```
enrichment Eq_Bool_Mod_Nat
operators
  =, /=: (bool, bool) -> bool
variables
  b1, b2: bool
equations
  1: true = true ==> true
  2: true = false ==> false
  3: false = true ==> false
  4: false = false ==> true
  5: b1 /= b2 ==> not(b1 = b2)
models
  eq_bool : Equality[bool operators =]
  eq_nat : Equality[nat operators =]
end Eq_Bool_Mod_Nat
```

Une première remarque s'impose concernant la surcharge d'opérateurs.

Les opérateurs d'égalité portent le même nom (=). Toutefois, celui qui est mentionné dans la déclaration du modèle *eq_bool* fait référence à l'opérateur qui est défini dans ce module *Eq_Bool_Mod_Nat*, alors que le deuxième, présent dans la déclaration du modèle *eq_nat* fait référence à l'opérateur spécifié dans le module *Natural* (la sorte de leurs arguments permet de les différencier).

Cet exemple déclare donc deux modèles différents de la même propriété *Equality*. La signature partielle du module *Eq_Bool_Mod_Nat* est

$$\Sigma_{Eq_Bool_Mod_Nat}=(<\emptyset, \emptyset>, <\emptyset, \{=, /=\}>, <\emptyset, \emptyset>).$$

Il y a deux possibilités, en LPG, de manipuler des instances de propriété.

La première consiste à donner explicitement le modèle qui nous intéresse lors de l'utilisation de l'objet générique. Cette méthode est décrite dans le paragraphe *Mécanisme d'instanciation.*

La deuxième possibilité consiste à déclarer, au niveau d'un module (*type* ou *enrichment*) un modèle de propriété. On établit en fait la correspondance entre les objets formels décrits dans la propriété dont on déclare un modèle et des objets effectifs définis dans divers modules, éventuellement le module dans lequel on fait la déclaration.

Soit U un module LPG (*type* ou *enrichment*) présenté par $P_U=(\Sigma_U,E_U)$ dans lequel on déclare un modèle de la propriété P présentée par $P_P=(\Sigma_P,E_P)$, alors $P \xrightarrow{m} U$ dénote une déclaration de modèle. Le morphisme associé est

$$P_P=(\Sigma_P,E_P) \xrightarrow{\Phi^m} P_U=(\Sigma_U,E_U).$$

Une déclaration de modèle, en LPG prend la forme générale suivante :

***models** P[S_1, S_2, ..., S_m **operators** F_1, F_2, ..., F_n **predicates** P_1, P_2, ..., P_q].*

On donne, après le mot-clé *models*, le nom de la propriété dont on déclare un modèle et ensuite les images par Φ^m_S (respectivement Φ^m_Ω et Φ^m_Π) des sortes (respectivement opérateurs et prédicats) de cette propriété *P*.

La sémantique associée au module cible de cette déclaration de modèle est donnée par le triplet (Th(P), Φ^m, Th(U)) où :

- Th(P) est la théorie présentée par P_P,
- Φ^m est le morphisme défini par la déclaration du modèle et
- Th(U) est la théorie présentée par P_U.

Explicitons les morphismes définis par les déclarations des modèles *eq_bool* et *eq_nat* sur notre exemple :
pour *eq_bool* :

- Φ^m_S : t→bool, bool→bool,
- Φ^m_Ω : =→=, true→true, false→false, not→not, and→and, or→or et
- Φ^m_Π : $\emptyset \rightarrow \emptyset$

pour *eq_nat* :

- Φ^m_S : t→nat, bool→bool,

- Φ^m_Ω : =→=, true→true, false→false, not→not, and→and, or→or et
- Φ^m_Π : Ø→Ø.

La signature de la présentation de la théorie cible reste, quant à elle, inchangée et est égale, en tenant compte des importations à

$$\Sigma_{Eq_Bool_Mod_Nat}=(<\emptyset, \{bool, nat\}>, <\emptyset, \{=, /=, true, false, not, and, or, 0, succ, +, *, =\}>, <\emptyset, \emptyset>).$$

5.5- Mécanisme d'instanciation

Le mécanisme d'instanciation consiste à associer un modèle d'une propriété P à un objet générique qui a été défini dans un module exigeant cette propriété P.

Voyons les deux façons existant en LPG de créer des instanciations.

Supposons qu'il existe un modèle unique de la propriété exigée (qui a pu être déclaré dans un module quelconque). Alors, l'emploi d'un objet générique défini dans un module qui exige cette propriété utilisera automatiquement le modèle existant. C'est l'instanciation implicite.

Si, par contre il n'existe pas de modèle de la propriété, ou bien si l'on désire utiliser l'objet générique avec un autre modèle que celui déclaré, il est possible de donner explicitement lors de l'utilisation de l'objet générique le modèle voulu. Dans ce deuxième cas, on parle d'instanciation explicite.

La relation introduite par le premier cas se rammène à une relation d'importation. Celle introduite par le deuxième cas est équivalente à une relation de déclaration de modèle.

On peut voir ce type de relation sur l'exemple suivant qui défini deux opérateurs réalisant le tri de séquences de naturels.

Le premier trie la séquence par ordre croissant des valeurs, le deuxième trie par ordre décroissant.

On donne par la même occasion la spécification d'un enrichissement destiné essentiellement à introduire un modèle de la propriété *Total_Order* sur les naturels. Cet enrichissement contient les définitions des opérateurs de différence et de comparaison entre naturels.

On donne également un enrichissment des séquences définissant une fonction de tri.

```
enrichment Nat_Ops
operators
  /=, <=, <, >=, > : (nat, nat) -> bool
variables
  n,m: nat
equations
```

```
  1: 0<=n ==> true
  2: succ(m)<=0 ==> false
  3: succ(m)<=succ(n) ==> m<=n
  4: m > n ==> not(m <=n)
  5: m >= n ==> n <= m
  6: m < n ==> not(m >= n)
  7: m /= n ==> not(m=n)
models
  inc_nat : Total_Order[nat operators <=, =]
end Nat_Ops
```

La signature de ce module est

$$\Sigma_{Nat_Ops}=(<\emptyset, \{nat, bool\}>, <\emptyset, \{/=, <=, <, >=, >, 0, succ, +, *, =, true, false, not, and, or\}>, <\emptyset, \emptyset>).$$

```
enrichissement Sorting
requires Total_Order[val operators infeq, eq]
operators
   sort : seq[val] -> seq[val]
   insert : (val, seq[val]) -> seq[val]
variables
   a, b: val
   s: seq[val]
equations
   1: sort(nil:seq[val]) ==> nil:seq[val]
   2: sort(a<+s) ==> insert(a, sort(s)
   3: insert(a, nil:seq[val]) ==> a<+nil:seq[val]
   4: insert(a, b<+s) ==>   if infeq(a, b) then
                               a<+(b<+s)
                            else
                               b<+insert(a, s)
                            endif
end Sorting
```

La signature de de module est

$$\Sigma_{Sorting}=(<\{val\}, \{val, seq\}>, <\{infeq, eq\}, \{infeq, eq, sort, insert, nil, <+\}>, <\emptyset, \emptyset>)$$

```
enrichment Nat_Sorting
  operators
    ns_inc : seq[nat] -> seq[nat]
    ns_dec : seq[nat] -> seq[nat]
  variables
    s : seq[nat]
  equations
    1 : ns_inc (s) ==> sort (s)
    2 : ns_dec (s) ==> sort.Total_Order[nat operators >=, =] (s)
```

end Nat_Sorting

La signature de ce module est

$$\Sigma_{Nat_Sorting}=(<\emptyset, \{seq, nat, bool\}>, <\emptyset, \{nil, <+, 0, succ, +, *, =, /=, <=, <, >=, >, true, false, not, and, or, ns_inc, ns_dec\}>, <\emptyset, \emptyset>).$$

Les deux opérateurs introduits dans *Nat_Sorting* ne font qu'un appel à l'opérateur *sort* qui exige une propriété d'ordre total sur les éléments de la séquence à trier.

La différence réside dans le mécanisme d'instanciation.

Pour *ns_inc*, l'opérateur *sort* est utilisé avec une instance implicite de la propriété *Total_Order* qui aura été déclaré dans un autre module.

L'opérateur *ns_dec* utilise le même opérateur *sort* cette fois-ci instancié explicitement. On donne, en effet, le modèle complet de la propriété *Total_Order* nécessaire au bon fonctionnement pour les comparaisons du tri.

Considérant la notion d'instanciation, il existe dans cet exemple trois relations, une importation et deux déclarations de modèle.

Le morphisme associé à l'importation (de *Nat_Ops* dans *Nat_Sorting*) est le suivant :

- Φ^i_S : bool→bool, nat→nat,
- Φ^i_Ω : true→true, false→false, not→not, and→and, or→or, 0→0, succ→succ, +→+, *→*, =→=, /=→/=, <=→<=, <→<, >=→>=, >→> et
- Φ^i_Π : $\emptyset\rightarrow\emptyset$.

Celui associé à la déclaration du modèle *seq[nat]* est :

- Φ^m_S : t→nat,
- Φ^m_Ω : $\emptyset\rightarrow\emptyset$ et
- Φ^m_Π : $\emptyset\rightarrow\emptyset$.

Celui associé à la déclaration du modèle *Total_Order[nat operators >=, =]* est :

- Φ^m_S : t→nat,
- Φ^m_Ω : <=→>=, =→= et
- Φ^m_Π : $\emptyset\rightarrow\emptyset$.

5.6- Clause *satisfies*

Une clause *satisfies* établit une ou plusieurs relations entre des propriétés.

Etant donnée une propriété P, une clause *satisfies* P_s présente dans cette propriété P introduit un morphisme de théorie. La validation de ce morphisme exige que les axiomes de la propriété P_s soient des théorèmes pour la propriété P.

On peut introduire ainsi une notion de vérification sémantique de la spécification de P.

Voici, en tant qu'exemple, les spécifications d'une relation d'égalité (propriété *Equality*) et d'une relation d'ordre partiel (propriété *Partial_Order*). Une troisième propriété, *Formal_Sort*, figure dans cet exemple. Elle introduit une sorte formelle et donc permet de décrire un ensemble de valeurs.

```
property Formal_Sort                        — t will be a formal sort
sorts t
end Formal_Sort

property Equality
sorts t
operators
   =: (t,t)->bool                    — The abstract data type of the Boolean
variables                            —   values is defined elsewhere
   x,y,z: t
equations
  1: x=x == true
  2: x=y == y=x
  3: (x=y and y=z) => x=z == true
satisfies
  Formal_Sort[t]
end Equality

property Partial_Order
sorts t
operators
  <=, =: (t,t) -> bool                      — <= will represent all partial order
relation
variables                                   —   the description with the equations
its
  x,y,z: t                                  —   behaviour needs the operator = whic
must
equations                                   —   be an equality operator
  1: x<=x == true                           —   reflexivity
  2: x=y == x<=y and y<=x                   —   antisymmetry
  3: (x<=y and y<=z) => x<=z == true        —   transitivity
satisfies
```

```
    Equality[t operators =]
end Partial_Order
```

Les signatures de ces modules sont respectivement

$$\Sigma_{Formal_Sort}=(<\emptyset, \{t\}>, <\emptyset, \emptyset>, <\emptyset, \emptyset>),$$

$$\Sigma_{Equality}=(<\emptyset, \{t, bool\}>, <\emptyset, \{=, true, false, not, and, or\}>,<\emptyset, \emptyset>) \text{ et}$$

$$\Sigma_{Partial_Order}(<\emptyset, \{t, bool\}>, <\emptyset, \{<=, =, true, false, not, and, or\}>, <\emptyset, \emptyset>).$$

Cet exemple, issu de la bibliothèque prédéfinie du système LPG, introduit par deux fois la clause *satisfies*.

La relation introduite par la clause *satisfies* de la propriété *Partial_Order* exprime le fait que l'opérateur = utilisé pour la spécification de l'anti-symétrie de <= doit se comporter comme un opérateur d'égalité. Une autre façon de dire est que, la spécification de *Partial_Order* suppose que l'opérateur symbolisé par = satisfait l'axiomatisation de l'égalité.

Il en va de même pour la clause *satisfies* présente dans la propriété *Equality*. Elle signifie que que la spécification de l'égalité qui utilise une sorte formelle quelconque, doit, pour être correcte, vérifier l'axiomatisation de la propriété *Formal_Sort*.

Vérifier que l'opérateur = de la propriété *Partial_Order* satisfait l'axiomatisation de l'égalité revient pratiquement à prouver que les axiomes de la propriété *Equality*, après le renommage défini dans la clause *satisfies* présente dans la propriété *Partial_Order*, sont des théorèmes de la théorie définie par la présentation de *Partial_Order*.

Plus généralement, une clause

$$\textbf{satisfies } P_s[S_{s_1}, S_{s_2}, ..., S_{s_m} \textbf{ operators } F_{s_1}, F_{s_2}, ..., F_{s_n} \textbf{ predicates } P_{s_1}, P_{s_2}, ..., P_{s_q}]$$

se trouvant dans une propriété P (présentée par $P_P=(\Sigma_P, E_P)$) indique que tout modèle de P_s (présentée par $P_{P_s}=(\Sigma_{P_s}, E_{P_s})$ peut être construit à partir d'un modèle de P.

Autrement dit, soit Φ^s le morphisme

$$P_{P_s}=(\Sigma_{P_s}, E_{P_s}) \xrightarrow{\Phi^s} P_P=(\Sigma_P,E_P),$$

cette relation traduit l'existence de l'application $\bar{\Phi}^s$:$Alg_{\Sigma_P} \rightarrow Alg_{\Sigma_{P_s}}$ qui déduit une Σ_{P_s}-algèbre A d'une Σ_P-algèbre A' en oubliant les ensembles et les fonctions de A' qui ne sont pas dans A.

Pratiquement, dès qu'un modèle de la propriété *Partial_Order* sera déclaré (par exemple *Partial_Order[nat* ***operators*** *<=, =]*), le système sera capable, du fait de la clause *satisfies* de la propriété *Partial_Order*, d'en déduire un modèle de la propriété *Equality* (par exemple *Equality[nat* ***operators*** *=]*) et, du fait de la clause

satisfies de la propriété *Equality*, un modèle de la propriété *Formal_Sort* (sur l'exemple, *Formal_Sort[nat]*).

Cette relation peut se noter $P_S \xrightarrow{s} P$ et la sémantique que l'on peut donner à une propriété P qui est la cible d'un tel morphisme est donnée par le triplet ($Th(P_S)$, Φ^s, Th(P)) où :

- $Th(P_S)$ est la théorie présentée par P_{P_S},
- Φ^s est le morphisme défini par la notation de la clause *satisfies* et
- Th(P) est la théorie présentée par P_P.

Regardons à quoi ressemble les morphismes introduits par les exemples précédents.

Le morphisme défini par la clause *satisfies* de la propriété *Equality* est :

- $\Phi^s_S : t \rightarrow t$,
- $\Phi^s_\Omega : \emptyset \rightarrow \emptyset$ et
- $\Phi^s_\Pi : \emptyset \rightarrow \emptyset$.

Celui défini par la clause *satisfies* de la propriété *Partial_Order* est :

- $\Phi^s_S : t \rightarrow t$,
- $\Phi^s_\Omega : = \rightarrow =$ et
- $\Phi^s_\Pi : \emptyset \rightarrow \emptyset$.

Dans chaque cas, la signature des modules cibles ($\Sigma_{Equality}$ et $\Sigma_{Partial_Order}$) n'est pas modifiée.

5.7- Clause *inherits*

Une clause *inherits* établit également une ou plusieurs relations entre des propriétés.

Intuitivement, elle permet de récupérer les définitions d'autres propriétés après renommage et de les inclure dans la propriété où elle figure.

Voici, par exemple, la spécification d'une relation d'ordre total (exemple de la bibliothèque prédéfinie du système LPG) qui est identique à celle de l'ordre partiel, vue précédemment, à laquelle on rajoute une équation exprimant que les éléments de tout couple de valeurs sont comparables.

```
property Total_Order
sorts t
operators
```

```
  <=, =: (t,t) -> bool                    — <= will represent all total order
variables                                 --   relation
  x,y: t
equations                                 — It may be defined with the same
  1: x<= y or y<=x == true                —  equations as those defining the
inherits                                  —  partial order relation and another
  Partial_Order[t operators <=,=]         —  one
end Total_Order
```

La signature de ce module est la suivante

Σ_{Total_Order}=(<Ø, {t, bool}>, <Ø, {<=, =, true, false, not, and, or}>, <Ø, Ø>).

Cette propriété *Total_Order* a la même sémantique que la propriété *Partial_Order* à laquelle on rajoute l'équation *x<= y or y<=x == true*. C'est la raison pour laquelle l'opérateur = figure dans la signature bien qu'il ne soit pas utilisé dans les équations propres à *Total_Order*.

Une clause

inherits $P_h[S_{h_1}, S_{h_2}, ..., S_{h_m}$ ***operators*** $F_{h_1}, F_{h_2}, ..., F_{h_n}$ ***predicates*** $P_{h_1}, P_{h_2}, ..., P_{h_q}]$

mentionnée dans une propriété P (présentée par $P_P=(\Sigma_P, E_P)$) signifie que les équations figurant dans la propriété P_h (présentée par $P_{P_h}=(\Sigma_{P_h}, E_{P_h})$) doivent être ajoutées à celles figurant dans P après substitution par les sortes S_{h_i}, les opérateurs F_{h_j} et les prédicats P_{h_k}.

De même que pour la clause *satisfies*, une déclaration de modèle d'une propriété P dans laquelle figure une clause *inherits* d'une propriété P_h génère automatiquement un modèle de la propriété P_h.

Sur l'exemple, la déclaration d'un modèle de *Total_Order* déclenchera la création d'un modèle de *Partial_Order*, qui génèrera les modèles d'autres propriétés comme on l'a vu précédemment.

On peut noter une telle relation $P_h \xrightarrow{h} P$. Elle correspond au morphisme Φ^h

$$P_{P_h}=(\Sigma_{P_h}, E_{P_h}) \xrightarrow{\Phi^h} P_P=(\Sigma_P, E_P).$$

La sémantique que l'on peut associer à une propriété qui est la cible d'une telle relation est donnée par le triplet ($Th(P_h)$, Φ^h, Th(P)) où :

- $Th(P_h)$ est la théorie présentée par P_{P_h},
- Φ^h est le morphisme défini dans la notation de la clause *inherits* et
- Th(P) est la théorie présentée par $(S_P, \Omega_P, \Pi_P, \Phi^h_E(E_{P_h}) \oplus E_P)$ sous

les conditions suivantes

- $\Phi_S(S_{P_h}) \subseteq S_P$, (1)

- $\Phi_\Omega(\Omega_{P_h}) \subseteq \Omega_P$ et (2)
- $\Phi_\Pi(\Pi_{P_h}) \subseteq \Pi_P$. (3)

On voit ainsi que la théorie associée à la propriété cible du morphisme Φ^h est construite à partir de la présentation partielle de cette propriété à laquelle on ajoute les équations renommées de la propriété source $\Phi^h_E(E_{P_h})$.

Les trois conditions expriment le fait que les sortes manipulées dans la propriété source du morphisme ϕ^h doivent être déclarée dans la propriété cible.

Pour notre exemple, le morphisme introduit est le suivant :

- Φ^h_S : t→t,
- Φ^h_Ω : <=→<=, =→= et
- Φ^h_Π : Ø→Ø.

La signature est identique à cause des trois conditions explicitées ci-dessus. Seules les équations de *Partial_Order*, après renommages, sont incluses dans la présentation P_{Total_Order}.

5.8- Clause *combines*

Intuitivement, la présence d'une clause *combines* dans une propriété *P* signifie que la propriété *P* où apparait cette clause représente la même spécification que la réunion des différentes propriétés apparaissant dans cette clause *combines*.

L'exemple suivant montre les spécifications concernant d'une part la commutativité d'un opérateur et d'autre part l'associativité.

A partir de ces deux decriptions indépendantes, on consruit une spécification décrivant la classe des opérateurs qui sont à la fois associatifs et commutatifs.

```
property Com
sorts t
operators
  +: (t,t)->t
variables
  x, y: t
equations
  1: x+y == y+x
satisfies
  Formal_Sort[t]
end Com
```

```
property Assoc
sorts t
operators
  +: (t,t)->t
variables
  x, y, z: t
equations
  1: x+(y+z) == (x+y) +z
satisfies
  Formal_Sort[t]
end Assoc

property Assoc_Com
sorts t
operators
  +: (t,t) -> t
combines
  Com[t operators +],
  Assoc[t operators +]
end Assoc_Com
```

Les signatures de ces modules sont

$$\Sigma_{Com}=(<\emptyset, \{t\}>, <\emptyset, \{+\}>, <\emptyset, \emptyset>)$$

pour la propriété *Com*,

$$\Sigma_{Assoc}=(<\emptyset, \{t\}>, <\emptyset, \{+\}>, <\emptyset, \emptyset>)$$

pour *Assoc* et

$$\Sigma_{Assoc_Com}=(<\emptyset, \{t\}>, <\emptyset, \{+\}>, <\emptyset, \emptyset>)$$

pour la propriété combinante *Assoc_Com.*

La clause *combines* établit donc des relations entre une ou plusieurs propriétés sources P^x et une propriété cible P. Elle s'écrit, en LPG

$$\textbf{combines } P_c^1[S_{c_1}^1, S_{c_2}^1, ..., S_{c_{m_1}}^1 \textbf{ operators } F_{c_1}^1, F_{c_2}^1, ..., F_{c_{n_1}}^1 \textbf{ predicates } P_{c_1}^1, P_{c_2}^1, ..., P_{c_{q_1}}^1],$$

$$P_c^2[S_{c_1}^2, S_{c_2}^2, ..., S_{c_{m_2}}^2 \textbf{ operators } F_{c_1}^2, F_{c_2}^2, ..., F_{c_{n_2}}^2 \textbf{ predicates } P_{c_1}^2, P_{c_2}^2, ..., P_{c_{q_2}}^2],$$

$$...,$$

$$P_c^r[S_{c_1}^r, S_{c_2}^r, ..., S_{c_{m_r}}^r \textbf{ operators } F_{c_1}^r, F_{c_2}^r, ..., F_{c_{n_r}}^r \textbf{ predicates } P_{c_1}^r, P_{c_2}^r, ..., P_{c_{q_r}}^r].$$

On donne après le mot clé *combines* la liste des propriétés avec le morphisme associé.

On peut dire que la propriété P (présentée par $P_P=(\Sigma_P, E_P)$) où est écrite une clause *combines* est équivalente à l'union des propriétés $P^{x,x=1,...,r}$ après

substitution des sortes, opérateurs et prédicats formels par les sortes, opérateurs et prédicats effectifs $\left(\left(S_{c^x_{i,i=1,\ldots,m_x}}, F_{c^x_{j,j=1,\ldots,n_x}}, P_{c^x_{k,k=1,\ldots,q_x}}\right), x=1,\ldots,r\right)$.

Une conséquence de cette définition est que la propriété P ne doit introduire ni sortes, ni opérateurs, ni prédicats supplémentaires et ne doit non plus définir ni équations, ni clauses.

Par ailleurs, les signatures de chacune des propriétés combinées $P^{x,x=1,\ldots,r}$ doivent recouvrir la signature de la propriété combinante P.

Plus formellement, l'ensemble des sortes (S_P), resp. des opérateurs (Ω_P) et des prédicats (Π_P), est égal à l'union ensembliste des ensembles des sortes ($S_{c^r_{m_r}}$), respectivement des opérateurs ($F_{c^r_{n_r}}$) et des prédicats ($P_{c^r_{q_r}}$), de chacune des propriétés combinées.

La déclaration d'un modèle de P génèrera un modèle pour chaque propriété P^r_c. Réciproquement, du fait de l'équivalence, dès que tous les modèles des propriétés P^r_c,sont créés, un modèle de la propriété P est également créé, de façon automatique.

Sur l'exemple, si un modèle de *Assoc_Com* est déclaré, automatiquement, les modèles de *Assoc* et de *Com* seront connus du système. A l'inverse, si le système connait un modèle de *Com* et un modèle de *Assoc*, il en déduira automatiquement un modèle de *Assoc_Com*.

Une telle clause présente dans une propriété *P* introduit *r* relations entre chacune des propriété $P^{x,x=1,\ldots,r}_c$ et la propriété *P*. Ces relations peuvent être dénotées $\left(P^x_c \xrightarrow{c} P\right)^{x=1,\ldots,r}$.

La sémantique que l'on peut donner à une propriété cible d'une telle liste de relations est donnée par r triplets : $\left(Th(P^x_c), \Phi^c_x, Th(P)\right)^{x=1,\ldots,r}$ où :

- $Th(P^{x,x=1,\ldots,r}_c)$ sont les théories présentées par $P_{P^{x,x=1,\ldots,r}_c}$,
- $\Phi^c_{x,x=1,\ldots,r}$ sont les morphismes existant entre $Th(P^{x,x=1,\ldots,r}_c)$ et Th(P), le $x^{ième}$ morphisme est défini par le $x^{ième}$ élément de la clause *combines* et
- Th(P) est la théorie présentée par :

$$\left(\bigcup_x\left(\Phi^c_{xS}(S_{P^x_c})\right)_{x=1,\ldots,r}, \bigcup_x\left(\Phi^c_{x\Omega}(\Omega_{P^x_c})\right)_{x=1,\ldots,r}, \bigcup_x\left(\Phi^c_{x\Pi}(\Pi_{P^x_c})\right)_{x=1,\ldots,r}, \bigcup_x\left(\Phi^c_{xE}(E_{P^x_c})\right)_{x=1,\ldots,r}\right)$$

Du fait de la sémantique de la clause combine, on doit avoir :

- $\bigcup_x \left(\Phi^c_{x\,S}(S_{P^x_c}) \right)_{x=1,\ldots,r} = S_P$,
- $\bigcup_x \left(\Phi^c_{x\,\Omega}(\Omega_{P^x_c}) \right)_{x=1,\ldots,r} = \Omega_P$ et
- $\bigcup_x \left(\Phi^c_{x\,\Pi}(\Pi_{P^x_c}) \right)_{x=1,\ldots,r} = \Pi_P$.

Ce qui exprime que l'union des différentes signatures renommées des propriétés sources doit être identique à la signature de la propriété cible.

Sur notre exemple, la clause combines introduit deux morphismes (Φ^c_1 et Φ^c_2). Le premier prend sa source dans la propriété *Com*, le deuxième dans Assoc :

- $\Phi^c_{1\,S} : t \rightarrow t$,
- $\Phi^c_{1\,\Omega} : + \rightarrow +$ et
- $\Phi^c_{1\,\Pi} : \emptyset \rightarrow \emptyset$.
- $\Phi^c_{2\,S} : t \rightarrow t$,
- $\Phi^c_{2\,\Omega} : + \rightarrow +$ et
- $\Phi^c_{2\,\Pi} : \emptyset \rightarrow \emptyset$.

Et la théorie Th(P) est présentée par

- $\bigcup_x \left(\Phi^c_{x\,S}(S_{P^x_c}) \right)_{x=1,2} = \langle \emptyset, \{t\} \rangle$,
- $\bigcup_x \left(\Phi^c_{x\,\Omega}(\Omega_{P^x_c}) \right)_{x=1,2} = \langle \emptyset, \{+\} \rangle$,
- $\bigcup_x \left(\Phi^c_{x\,\Pi}(\Pi_{P^x_c}) \right)_{x=1,2} = \langle \emptyset, \emptyset \rangle$ et
- $\bigcup_x \left(\Phi^c_{x\,E}(E_{P^x_c}) \right)_{x=1,2} = \{x+y == y+x,\ x+(y+z) == (x+y)+z\}$.

6- Interface

6.1- Introduction

Une spécification algébrique peut s'écrire à l'aide de plusieurs modules. Chaque module peut être en relation avec un ou plusieurs autres. Cette possibilité de structuration nécessite de vérifier certaines conditions afin que le spécification algébrique globale ne contienne pas d'incohérence.

Nous venons de voir que les spécifications ou programmes LPG peuvent être structurés les uns par rapport aux autres en utilisant cette notion de module. On peut considérer chacun de ces modules comme une théorie partiellement présentée par les définitions intrinsèques à ce module. Toutes ces théories peuvent être liées entre elles par des morphismes.

Il est de ce fait nécessaire de prouver que nous avons bien affaire à des morphismes de théories. En effet, si $\Phi : (\Sigma_s, E_s) \rightarrow (\Sigma_t, E_t)$ est un morphisme de théorie, alors on doit avoir $\forall e{:}t_1 == t_2 \in E_s$, $\hat{\Phi}_\Sigma(t_1) == \hat{\Phi}_\Sigma(t_2) \in \overset{\circ}{E}_t$. En d'autres termes, si $t_1 == t_2$ est un axiome de la théorie source, $\hat{\Phi}_\Sigma(t_1) == \hat{\Phi}_\Sigma(t_2)$ doit être une conséquence logique (un théorème) de la théorie cible.

Nous avons tout d'abord présenté un système permettant de d'assiter un utilisateur dans l'écriture de démonstrations. Nous avons alors présenté le langage LPG qui implémente sous forme de modules la notion de théories structurées et nous avons formalisé les différentes relations pouvant exister entre ces modules. Ce chapitre présente maintenant la notion de validation sémantique.

On introduit dans un premier temps la notion de validation sémantique. On passe pour cela en revue les différentes relations étudiées dans le chapitre précédent. Pour chacune de ces relations, on donne les conditions qui doivent être vérifiées.

On explicite ensuite la notion de théorie structurée en la décrivant dans le contexte du système LPG ainsi que de celui de LCF, qui est présenté brièvement. On peut trouver, en effet, dans [SB 83] un travail relatif à l'écriture et la manipulation de théories structurées à l'aide du système LCF.

On définit la notion de présentation étendue qui sera utilisée pour réaliser les démonstrations ainsi que la manière de retrouver les formules logiques qui doivent être démontrées.

On décrit le logiciel qui est issu de cette étude. On explique pour cela son fonctionnement et on décrit les diverses commandes qu'il offre. Ce logiciel se compose d'une partie qui permet de transférer des informations entre les systèmes LPG et OASIS (envoi de commandes et d'équations de LPG vers OASIS, récupération des résultats de OASIS par LPG). La deuxième partie de ce logiciel est chargée de construire l'ensemble des axiomes nécessaires pour réaliser les

démonstrations, ainsi que l'ensemble des formules qui doivent être démontrées afin de pouvoir considérer une spécification sémantiquement valide.

On donne enfin deux exemples d'utilisation de ce logiciel. Le premier consiste tout simplement à démontrer des formules logiques présentes dans la rubrique *theorems* d'une déclaration de type LPG. La connexion des systèmes LPG et OASIS a permis en effet, par effet de bord, de doter le système LPG de la possibilité de démontrer de telles formules logiques. Le deuxième exemple consiste à valider une spécification LPG.

6.2- Notion de validation sémantique

6.2.1- Morphismes valides par définition

Un module LPG sera considéré comme valide sémantiquement lorsque tous les morphismes de théories qui ont la théorie associée à ce module comme cible seront eux-mêmes valides.

Il faut remarquer que les morphismes associés à certaines relations sont valides par construction. C'est le cas des relations d'importation, de paramétrisation et celles introduites par les clauses *inherits* et *combines*.

En effet, l'importation entre un module U_s présenté par $P_{U_s}=(\Sigma_{U_s},E_{U_s})$ et un autre module U_t présenté par $\dot{P}_{U_t}=(\dot{\Sigma}_{U_t},\dot{E}_{U_t})$ est formalisée par le morphisme identité entre les théories $Th(U_s)$ et $Th(U_t)$. Il faudrait donc vérifier que $\forall e_s \in E_{U_s}$ ($Th(U_s)$ est la théorie présentée par (Σ_{U_s},E_{U_s})), $\Phi_E(e_s)=e_s \in Th(U_t)$, ce qui, par définition de la construction est vérifié car $Th(U_t)$ est la théorie présentée par

$$(\Sigma_{U_s} \oplus \dot{\Sigma}_{U_t},\ E_{U_s} \oplus \dot{E}_{U_t})$$

et donc $e_s \in E_{U_s} \oplus \dot{E}_{U_t}$.

Le cas concernant la relation de paramétrisation est relativement semblable. La différence vient du fait que le morphisme n'est pas un morphisme d'identité, il est explicité par la notation qui déclare la paramétrisation. Considérons un module U présenté par $\dot{P}_U=(\dot{\Sigma}_U,\dot{E}_U)$ qui exige une propriété P présentée par $P_P=(\Sigma_P,E_P)$. Le morphisme va de Th(P) dans Th(U). Il faut donc s'assurer que $\forall e_P \in Th(P)$, $\Phi^r_E(e_P) \in Th(U)$. Par définition de ce type de relation, cette condition est vérifiée. En effet, Th(U) est présentée par

$$(\Phi^r_\Sigma(\Sigma_P) \oplus \dot{\Sigma}_U,\ \Phi^r_E(E_P) \oplus \dot{E}_U)$$

et donc $\Phi^r_E(e_P) \in \Phi^r_E(E_P) \oplus \dot{E}_U$.

Le cas des clauses *inherits* et *combines* sont identiques à celui de la paramétrisation. Soit une propriété P présentée par $P_P=(\Sigma_P, E_P)$, si elle hérite des équations d'une autre propriété P_h présentée par $P_{P_h}=(\Sigma_{P_h}, E_{P_h})$, nous avons le morphisme entre $Th(P_h)$, qui est la théorie présentée par P_{P_h} et Th(P), qui est la théorie présentée par

$$(\Sigma_P, \Phi^h_E(E_{P_h}) \oplus E_P)$$

et donc $\Phi^h_E(e_{P_h}) \in \Phi^h_E(E_{P_h}) \oplus E_P$.

Si, maintenant, la propriété P combine plusieurs propriétés $P_c^{x,x=1,\dots,r}$, présentées respectivement par $P_{P_c^{x,x=1,\dots,r}}=\left(\Sigma_{P_c^{x,x=1,\dots,r}}, E_{P_c^{x,x=1,\dots,r}}\right)$, il y a en fait *r* morphismes à vérifier, partant de $Th(P_c^{x,x=1,\dots,r})$ et arrivant dans Th(P) qui est présentée par

$$\left(\bigcup_x\left(\Phi^c_{x\Sigma}(\Sigma_{P_c^x})\right)_{x=1,\dots,r}, \bigcup_x\left(\Phi^c_{xE}(E_{P_c^x})\right)_{x=1,\dots,r}\right)$$

et par conséquent $\left(e_{P_c^x} \in E_{P_c^x}, \Phi^c_{xE}(E_{P_c^x})\right)_{x=1,\dots,r} \in \bigcup_{x'}\left(\Phi^c_{x'\Sigma}(\Sigma_{P_c^{x'}})\right)_{x'=1,\dots,r}$.

6.2.2- Morphismes à vérifier

Les déclarations de modèles et la relation issue de la clause *satisfies*, introduisent des morphismes qui demandent quant à eux à être vérifiés.

Pour la déclaration d'un modèle de la propriété P, présentée par $P_P=(\Sigma_P,E_P)$, dans un module U, présenté par $P_U=(\Sigma_U,E_U)$, le morphisme est établit entre Th(P),théorie présentée par P_P, et Th(U), théorie présentée par P_U.

Il est alors nécessaire de vérifier que $\forall e \in E_P$, $\Phi^m(e_m) \in Th(U)$.

Le raisonnement quant à la clause *satisfies* est analogue. Soit une propriété P présentée par $P_P=(\Sigma_P, E_P)$ qui satisfait une propriété P_s présentée par $P_{P_s}=(S_{P_s}, E_{P_s})$, le morphisme part de $Th(P_s)$, théorie présentée par P_{P_s}, et arrive dans Th(P), théorie présentée par P_P.

On doit vérifier que $\forall e \in E_{P_s}$, $\Phi^s(e_{P_s}) \in Th(P)$.

6.2.3- Autres morphismes

Le dernier type de relation concerne la relation introduite par une instanciation. Nous avons vu que ce cas, en fait, est équivalent soit à une importation si on est en présence d'une instanciation implicite, soit à une déclaration de modèle si on est présence d'une instanciation explicite.

Pour une instanciation implicite, les conditions sont vérifiées par définition.

Une instanciation explicite, par contre, nécessite de vérifier que l'on a effectivement un morphisme entre la propriété dont on déclare un modèle et la théorie associée au module où est déclarée cette instanciation.

6.2.4- Processus de validation

D'une manière générale, la validation d'un module LPG revient à démontrer que certaines égalités entre termes, présentes dans les théories sources de morphismes, sont des théorèmes dans les théories cibles.

L'élaboration des spécifications étant structurée par les différentes relations vues précédemment, dans un premier temps, il faut s'assurer que les modules sources de morphismes sont eux-même valides. Si le module U_s est en relation avec le module U_t, ce qui est noté $U_s \xrightarrow{rel} U_t$, le module U_t ne pourra être validé que si le module U_s est lui-même validé.

Dans un deuxième temps, on vérifie que les équations de U_s sont des théorèmes pour la théorie associée à U_t.

Le processus de validation d'un module LPG se décompose ainsi en deux étapes principales :

- validation de tous les modules se trouvant dans le graphe des relations en amont du module en cours de validation,
- vérification de la validité de certains morphismes.

6.3- Les théories structurées

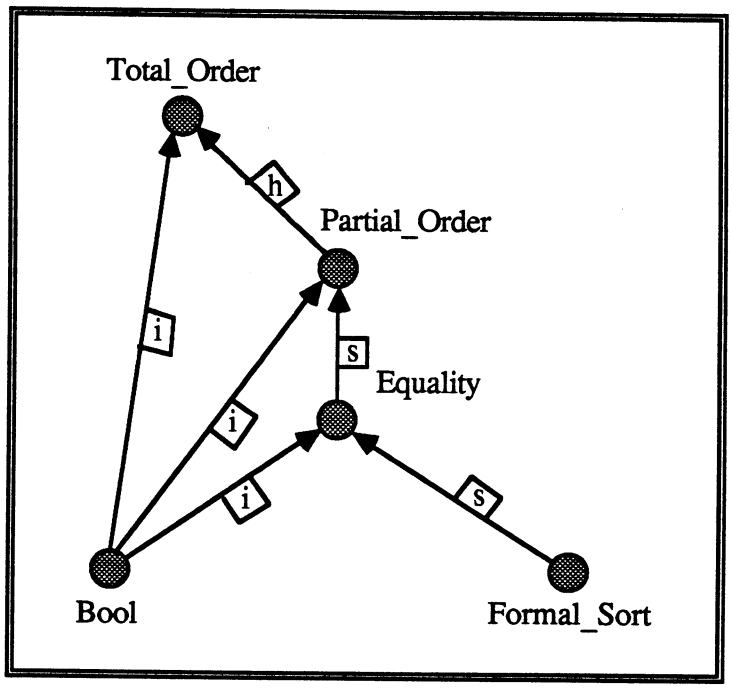
6.3.1- L'atelier LPG

On peut voir cette notion de structuration des modules LPG sur un exemple.

Considérons la propriété *Total_Order* dont la spécification a déjà été donnée. Elle importe des définitions de la spécification *Bool* (la sorte *bool* et les opérateurs *or* et *true*) et hérite des équations de la propriété *Partial_Order*.

Cette dernière importe également des définitions du module *Bool* (la sorte *bool* et les opérateurs *true*, *and* et =>) et doit satisfaire l'axiomatisation du module *Equality*.

Le module *Equality* importe aussi des définitions de *Bool* (la sorte *bool* et les opérateurs *true*, *and* et =>) et satisfait l'axiomatisation de *Formal_Sort*. On peut visualiser ces différentes relations sur le graphe suivant :



Relations concernant la propriété Total_Order

Chaque nœud de ce graphe représente un module LPG et les flèches indiquent la nature de la relation existant entre les modules (*s* par *satisfies*, *h* pour *inherits* et *i* pour importation).

A chacune des relations on peut faire correspondre, comme on l'a déjà vu, un morphisme de théorie.

Il faut remarquer que, afin de limiter la complexité, ce formalisme de description n'est pas tout à fait correct. Pour une description plus précise, on peut se référer à [Rey 87].

Cette structuration permet de simplifier les différentes démonstrations que l'on peut être amené à faire en LPG. En effet, les présentations sont localisées dans les différents nœuds du graphe et, étant donné une égalité à démontrer, on peut savoir de quels nœuds on aura besoin et par là même de limiter la quantité d'information (nombre d'axiome) à manipuler.

Pratiquement, les diverses démonstrations seront réalisées en utilisant l'outil OASIS. Le fonctionnement de ce dernier va donc imposer dans une certaine mesure la démarche à suivre pour conduire les preuves.

Le point le plus important concerne les présentations qu'il faut manipuler tout au long de la preuve. Le système OASIS permet de structurer ses propres spécifications grâce à l'importation explicite et à une ébauche de paramétrisation.

Cette structuration n'est toutefois que syntaxique. Dès qu'il faut se servir de la réécriture (afin de conduire une preuve, faire des tests de spécification, etc...) cette structuration disparait pour donner naissance à un système de réécriture monolithique. Ce dernier contiendra toutes les règles de la spécification dans

laquelle on réalise la preuve ainsi que celles se trouvant dans les différents objets OASIS importés (et ceci, par fermeture transitive) ou passés en paramètre.

Cette structuration, visible au niveau syntaxique, se trouve ainsi *aplatie*, tous les axiomes sont ramenés au même niveau. Le démonstrateur manipule ainsi tous les axiomes sans différenciation.

Il est nécessaire, de ce fait, dans le cadre de LPG, de constituer un système de réécriture dans sa totalité, correspondant à la théorie dans laquelle on veut faire les démonstrations, afin de le fournir au démonstrateur.

On perd certains avantages de la structuration des modules LPG. Etant donnée une formule à démontrer, on pourrait se servir de cette structuration afin de démontrer chaque sous-but dans un nœud particulier du graphe et n'utiliser les informations contenues dans les autres nœuds que lorsque c'est nécessaire. L'avantage que l'on peut néanmoins retirer de cette structuration est qu'elle permet de minimiser le nombre des axiomes que le démonstrateur devra manipuler.

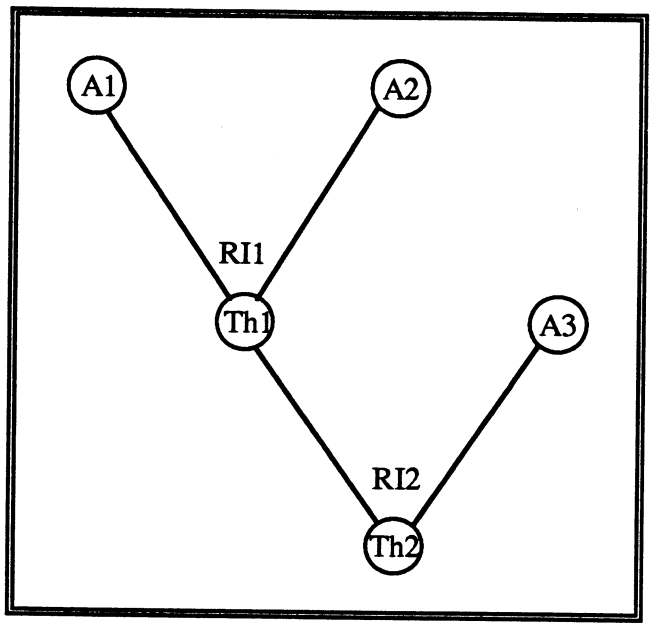
Une autre approche, celle proposée dans [SB 83], pour gérer l'information est possible. Cette approche manipule des théories structurées et conserve, pour réaliser les démonstrations, cette structuration.

6.3.2- Le système LCF

LCF est l'acronyme pour *Logic for Computable Functions*. Cette logique est due à Dana Scott et permet d'exprimer et de prouver des faits concernant les fonctions définies récursivement. C'est donc un système permettant de réaliser des preuves.

La logique de Scott est constituée d'un ensemble de phrases permettant d'écrire des assertions, d'un ensemble d'axiomes (phrases qui sont supposées être vraies) et d'un ensemble de règles d'inférence offrant la possibilité de générer de nouveaux théorèmes à partir des axiomes et des théorèmes précédemment démontrés.

Une preuve dans ce système peut se représenter comme une arborescence dont les feuilles sont des axiomes et les autres nœuds sont des théorèmes générés à partir de leurs propres fils grâce aux différentes règles d'inférence.



Arbre de preuve en LCF

Cet arbre de preuve représente le fait que *Th1* est un théorème issu des axiomes *A1* et *A2* par la règle d'inférence *RI1* et que le théorème *Th2* provient de *Th1* et de *A3* par la règle *RI2*.

Le premier système LCF a été développé à Stanford. Son but était de pouvoir construire de telles preuves. La méthode utilisée était celle qualifiée de *preuve conduite en arrière*. On part en effet d'une formule à démontrer et on construit l'arborescence jusqu'à obtenir des axiomes au niveau des feuilles. On disposait pour cela d'un ensemble fixe de commandes (*assume* pour désigner un nouveau but, *try* pour essayer une règle d'inférence, etc...).

Ce premier système a permis essentiellement de constater que la logique de Scott est assez puissante pour exprimer des problèmes compliqués. Cependant, la construction de preuve en donnant explicitement toutes les règles d'inférence s'avère fastidieuse. Il arrive assez fréquemment de répéter plusieurs fois, pour une même démonstration, des séquences identiques de règles d'inférence.

Une autre version de ce système LCF a été développée à Edinburgh [GMW 79]. Ce projet fut commencé en 1973 sous la responsabilité de Mr Milner. Il reprend les idées du système LCF de Stanford en supprimant le côté fastidieux du déroulement de la preuve. Il utilise pour cela un langage de programmation (en fait, le langage ML) dans lequel sont exprimées les commandes manipulant les preuves.

On peut ainsi définir des procédures regroupant l'application de règles d'inférence une bonne fois pour toute et, ensuite, utiliser ces procédures à volonté. La possibilité de définir de telles stratégies de preuve permet également d'avoir un ensemble non figé de commandes pour mener à bien une démonstration. De plus, le langage choisi pour manipuler les démonstrations (ML) est fortement typé, ce qui permet de différencier aisément les phrases générales que l'on peut construire des théorèmes (les théorèmes constituent un type abstrait particulier du système dont les différentes valeurs ne peuvent être construites qu'à partir de règles d'inférence).

Les preuves réalisées en LCF peuvent être conduites en avant. Mais on peut aussi réaliser des démonstrations dirigées par le but.

Une *tactique*, en LCF, permet d'obtenir une liste de sous-buts à partir d'un but, ainsi qu'une *validation*. On peut voir une tactique comme l'utilisation d'une règle d'inférence du but vers les prémisses.

La règle d'inférence $\frac{Th_1\ Th_2\ \ldots\ Th_n}{Th}$ indique que si on a démontré les théorèmes $Th_{i,\ i=1,2,\ldots,n}$ alors on peut déduire le nouveau théorème Th.

Regardant cette règle comme une tactique, on peut dire que Th sera un théorème si $Th_{i,\ i=1,\ldots,n}$ sont eux-mêmes des théorèmes, ce qui revient à essayer de démontrer chaque Th_i. La validation retournée par une tactique est nécessaire pour créer effectivement un nouveau théorème.

En effet, le seul moyen, en LCF, d'obtenir un nouveau théorème est d'appliquer des règles d'inférence sur des axiomes ou des théorèmes précédemment démontrés. La décomposition de but en sous-buts n'engendre donc pas de théorèmes, mais une validation, qui n'est rien d'autre qu'une fonction qui enregistre en quelque sorte la façon dont on pourra générer le théorème en utilisant les règles d'inférence. Il suffit d'évaluer les différentes validations obtenues dès que les listes de sous-buts sont vides.

Il est possible de combiner des tactiques entre elles en utilisant des *tacticielles*[1]. Une tacticielle prend ainsi une ou plusieurs tactiques et retourne un nouvelle tactique.

On peut avoir comme exemples de tacticielles :

- THEN : tactic x tactic → tactic, qui permet de combiner séquentiellement deux tactiques (la deuxième tactique est appliquée aux sous-buts délivrés par la première,
- ORELSE : tactic x tactic → tactic, qui exprime une combinaison alternative de deux tactiques (si la première échoue, la deuxième est appliquée),
- REPEAT : tactic → tactic, qui applique une tactique itérativement sur le but et les sous-buts produits tant que c'est possible (c'est à dire en s'arrêtant juste avant l'obtention d'un échec).

A partir de tactiques élémentaires, une tacticielle permet de construire des tactiques plus ou moins compliquées. Une *stratégie* en LCF désigne alors une telle tactique. On donne un but à une stratégie et elle renvoie une liste de sous-buts (éventuellement vide) et une validation. Si la liste est vide, l'évaluation de la validation transforme le but en un nouveau théorème.

On peut trouver dans [GMW 79] plusieurs exemples de tactiques, de tacticielles et de stratégies.

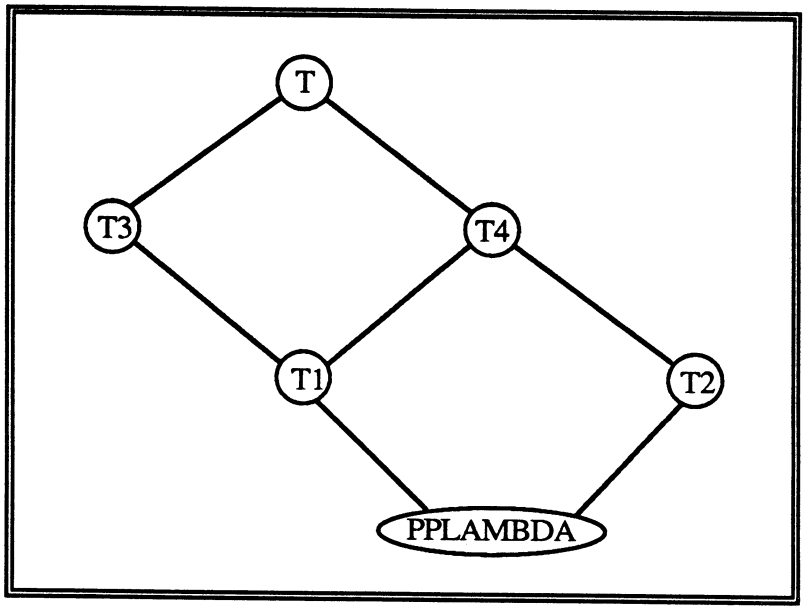
---

[1] *Tacticielle* est une traduction proposée pour le terme anglais *tactical*. Ce dernier a été créé à partir de *tactic* et par analogie avec l'extension entre *function* et *functional*.

LCF a été conçu de façon à pouvoir réaliser des démonstrations d'une manière naturelle. On y trouve la possibilité de faire de la déduction naturelle en utilisant les règles d'inférence. On peut faire des preuves dirigées par le but par l'intermédiaire des tactiques, tacticielles et stratégies. Ces différentes preuves sont réalisées dans des théories qui sont structurées les unes par rapport aux autres.

La structuration des théories est considérée, en LCF, comme une démarche naturelle pour réaliser des démonstrations. Cela se traduit par le fait que en débutant une session LCF, le système demande en tout premier lieu si on veut se placer dans une théorie existante ou bien créer une nouvelle théorie. Dans ce dernier cas, l'utilisateur doit indiquer quels sont les parents de la nouvelle théorie afin de la rattacher à celles précédemment créées.

De ce fait, toutes les théories présentes en LCF ont un ancêtre commun, la théorie appelée PPLAMBDA qui fait partie initialement du système.



Structuration des théories en LCF

Sur cet exemple de structuration, la théorie *T* a été créée en étendant les théories *T3* et *T4*. Il en va de même pour *T4* qui étend *T1* et *T2*. *T3* n'a été formée qu'à partir de *T1*. *T1* et *T2* sont issues directement de la théorie *PPLAMBDA*.

D'un point de vue structurel, le système LCF est constitué de trois parties.

ML (*MetaLanguage*) est un langage permettant, dans notre cas, de manipuler des preuves. Le deuxième élément est une logique de calcul appelée PPLAMBDA (***P**olymorphic **P**redicate* λ*-calculus*) et la troisième partie est une méthodologie de preuve dirigée par le but.

ML est un langage applicatif se rapprochant par divers aspects de ISWIM, POP-2 ou GEDANKEN. On peut déclarer et évaluer des expressions. Il est fortement typé, les types pouvant être polymorphes. A chaque expression, un contrôleur de type associe un type le plus général. Il permet de manipuler des objets

de types classiques (entiers, booléens, chaînes, caractères et les listes) ainsi que des objets de type particulier tels les termes (*term*), les formules (*form*) et les théorèmes (*thm*). On peut, de plus, déclarer et utiliser des types récursifs, des types abstraits. Le système de calcul de types permet de vérifier qu'un objet de type théorème est soit un axiome, soit obtenu par une règle d'inférence. On peut donc assurer la validité des théorèmes prouvés.

ML possède tous les avantages des langages fortement typés (contrôle sur les types) et, grâce au système de calcul de types, il offre la souplesse des langages non typés (on n'est pas obligé de donner explicitement le type des expressions, puisqu'il est calculé automatiquement). C'est un langage d'ordre supérieur. Les fonctions peuvent avoir elles-même des fonctions comme paramètres et retourner des fonctions comme résultats.

Une tactique, par exemple, est une fonction dont le profil est *but→(liste[but] x validation). validation* est elle-même une fonction de profil *liste[théo]→théo*. On voit déjà sur cet exemple qu'une tactique retourne, entre autre chose, une fonction comme résultat.

Une tacticielle qui n'est rien d'autre qu'un combinateur entre tactiques est un autre exemple de fonction qui prend une ou plusieurs fonction en paramètre (des tactiques) et retourne une fonction en résultat (également une tactique).

ML possède enfin un mécanisme d'exception et de récupération de ces exceptions qui s'avère particulièrement utile pour la programmation des stratégies (il permet de détecter automatiquement l'échec d'une stratégie de preuve et d'essayer une autre stratégie).

On peut voir dans [Gor 81] un exemple simple d'utilisation de ML pour représenter une logique particulière et réaliser des démonstrations dans cette logique.

PPLAMBDA [Pau 83b], [MMN 75] est une logique de calcul permettant de faire de la déduction naturelle. Deux classes d'objets y sont manipulés : les termes, qui représentent des valeurs calculables, et les formules qui sont des phrases logiques.

Les termes sont en fait ceux du λ-calcul typé. C'est à dire que les λ-expressions sont étiquetées par le type de la fonction qu'elles représentent. Par exemple, si la variable x est de type α et le terme t est de type β, la λ-expression $\lambda x.t$ est de type $\alpha \rightarrow \beta$.

Les formules atomiques sont construites à partir des λ-termes de la manière suivante. Si t_1 et t_2 sont deux λ-termes, alors $t_1 == t_2$ et $t_1 << t_2$ sont des formules atomiques. Viennent alors les formules composées qui sont construites à partir des formules atomiques et en utilisant des connecteurs logiques *conjonction* ($\wedge$), *implication* ($\Rightarrow$) et *quantification universelle* ($\forall$).

Si t_1 est de type $\alpha_1 \rightarrow \beta_1$ et t_2 est de type $\alpha_2 \rightarrow \beta_2$, alors le symbole $==$ exprime la relation d'égalité sur les domaines de définition des λ-termes ($t_1 == t_2$

signifie que le domaine de définition de t_1 est le même que celui de t_2 ($\alpha_1=\alpha_2$) et que $\forall x \in \alpha_1,\ t_1(x)=t_2(x)$). Le symbole $<<$ traduit une relation d'ordre partiel complet sur les domaines de définition des λ-expressions ($t_1<<t_2$ signifie que t_1 est moins défini que t_2, t_1 est défini sur α_1, t_2 est défini sur α_2 et on a $\alpha_1 \subseteq \alpha_2$ avec la condition $\forall x \in \alpha_1,\ t_1(x)=t_2(x)$).

On peut considérer PPLAMBDA comme une famille de calculs. Un élément de cette famille peut être défini par un triplet *(T, C, A)* où *T* est un ensemble d'opérateurs de types, *C* un ensemble de constantes et *A* un ensemble d'axiomes.

Il existe une famille minimale $F_0=(T_0, C_0, A_0)$ que l'on peut étendre en ajoutant des définitions.

T_0 contient des opérateurs sur les types comme *prod* (produit cartésien de deux types) ou *sum* (union disjointe).

C_0 contient, par exemple, *TT:tr* et *FF:tr* (les deux valeurs logiques, *e:type* est une notation pour désigner le type de l'expression *e*, ici, *tr* se rapporte aux valeurs booléennes), *FST:(#1,#2)→#1* et *SND:(#1,#2)→#2* (pour les projections du produit cartésien). Ces constantes sont formées d'une part du nom de la constante (ex : *FST*)et d'autre part de son type (ex : *(#1,#2)→#1*) éventuellement générique (pour LCF, le type sera générique si il existe un ou plusieurs types formels dans sa définition. *#1*, par exemple, est un type formel).

Pour des raisons pragmatiques, les axiomes contenus dans A_0 sont considérés comme des règles d'inférence, et leur utilisation est semblable à celle des fonctions. Comme exemple de règle d'inférence, on peut citer la conjonction $\frac{A_1]\text{-}w_1,\ A_2]\text{-}w_2}{A_1 \oplus A_2]\text{-}w_1 \wedge w_2}$ où la notation $A]\text{-}w$ représente un théorème composé d'une part d'un ensemble d'assertions *A* et d'autre part d'une formule bien formée *w*, $\oplus$ représente l'union ensembliste des assertions et w_1 et w_2 sont des formules bien formées. Cette règle exprime le fait que si on dispose des théorèmes $A_1]\text{-}w_1$ et $A_2]\text{-}w_2$, alors on peut déduire le théorème $A_1 \oplus A_2]\text{-}w_1 \wedge w_2$ où A_1 et A_2 sont des ensembles d'assertions (hypothèses),.

Cette famille F_0 peut être étendue en lui ajoutant d'autres définitions. On peut ainsi obtenir une nouvelle famille de calculs $F=(T, C, A)=F_0 \mathcal{U} F'$, où $F'=(T', C', A')$ et $\mathcal{U}$ est l'union entre familles définie par l'union disjointe entre les trois composantes respectives de chaque famille. Cette nouvelle famille *F* peut également être étendue à son tour et ainsi de suite. Les différentes familles de calcul sont ainsi ordonnées les unes par rapport aux autres ($F \overset{3}{\geq} F'$ si les constituants du triplet de F' sont inclus dans les constituants respectifs du triplet de *F*). La famille F_0 mentionnée plus haut est minimale par rapport à cette relation.

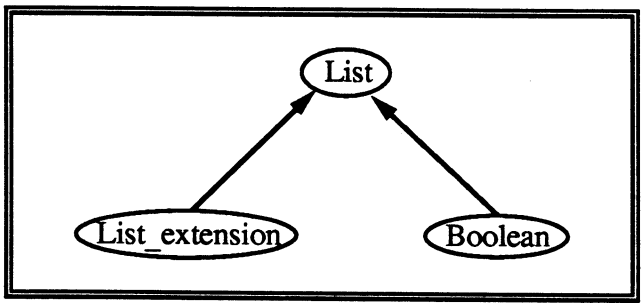
Faire une démonstration dans ce système LCF consiste à se placer dans une théorie particulière, existante ou non, puis à utiliser des stratégies ou des tactiques et enfin à appliquer les différentes fonctions de validation précédemment obtenues pour générer le théorème qui nous intéresse. Ceci implique d'avoir à sa disposition

les différents théorèmes et axiomes nécessaires pour la démonstration. Les axiomes ou théorèmes déjà démontrés, si ils ne figurent pas dans la théorie en cours, doivent pouvoir être extraits des théories ancestrales de la théorie courante.

La notion de structuration des théories a été approfondie dans [SB 83]. On peut considérer, par exemple, la théorie des listes :

```
List = extension of Boolean by :
types   list[t]
constants   nil : list[t]
            cons : t x list[t] -> list[t]
            head : list[t] -> t
            tail : list[t] -> list[t]
            null : list[t] -> bool
axioms  head(cons(x, l)) = x
        tail(cons(x, l)) = l
        null(cons(x, l)) = false
        null(nil) = true
```

Cette théorie peut être vue comme étant composée de la théorie des booléens[1] enrichie par l'extension décrite ci-dessus. Dans le cadre de LCF, une extension introduite de nouvelles définitions de types et d'opérateurs. Cette notion d'enrichissement peut se schématiser de la manière suivante :



Théorie des Listes en LCF

De la même manière, on peut vouloir définir la théorie des piles en LCF.

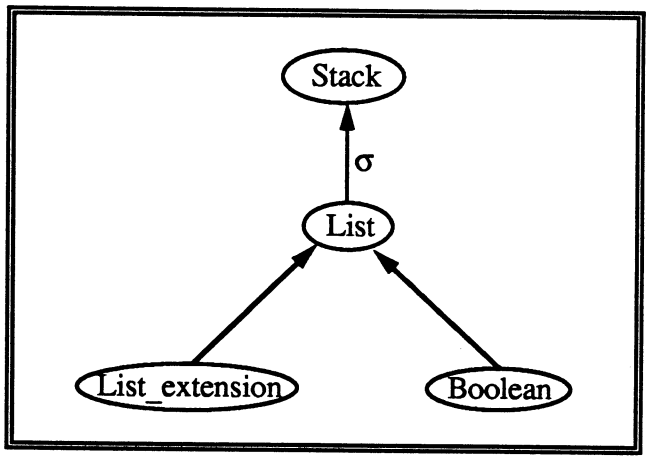
```
Stack = extension of Boolean by :
types   stack[t]
constants   nilstack : stack[t]
            push : t x stack[t] -> stack[t]
            top : stack[t] -> t
            pop : stack[t] -> stack[t]
            isempty : stack[t] -> bool
axioms  top(push(x, s)) = x
```

[1]On ne décrit pas ici la théorie *Boolean*.

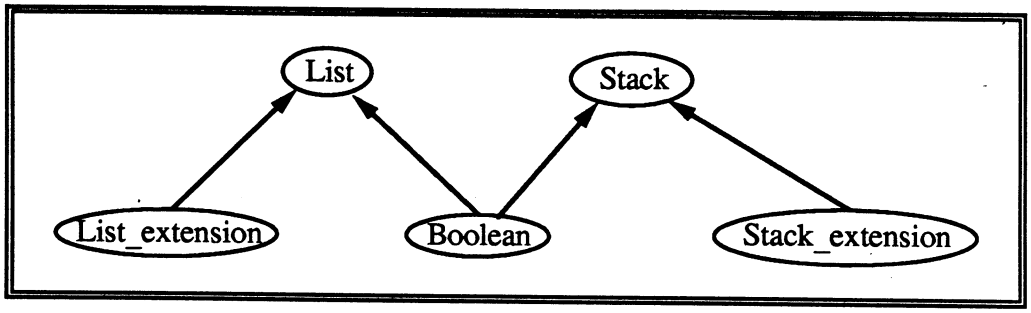
```
pop(push(x, s)) = s
isempty(push(x, s)) = false
isempty(nilstack) = true
```

On peut immédiatement remarquer l'analogie entre la théorie des listes et celle des piles. Plutôt que d'avoir à différencier les deux théories, on aimerait exprimer le fait que la théorie des piles est la même que la théorie des listes à un renommage σ près (les constantes et opérateurs de la théorie des booléens sont renommés en eux-même et ne figurent pas dans la description de σ) : σ=({list[t]→stack[t]}, {nil→nilstack, cons→push, head→top, tail→pop, null→isempty}).

Le principal avantage de cette structuration est que, à partir de tout théorème relatif à des listes, on peut déduire un théorème relatif aux piles. On obtient ainsi une structure plus simple.



Théorie LCF utilisant le renommage



Théorie LCF n'utilisant pas le renommage

Une autre relation envisagée est la notion d'image inverse. On peut voir cette notion sur l'exemple de la théorie des tables de symboles utilisées dans les

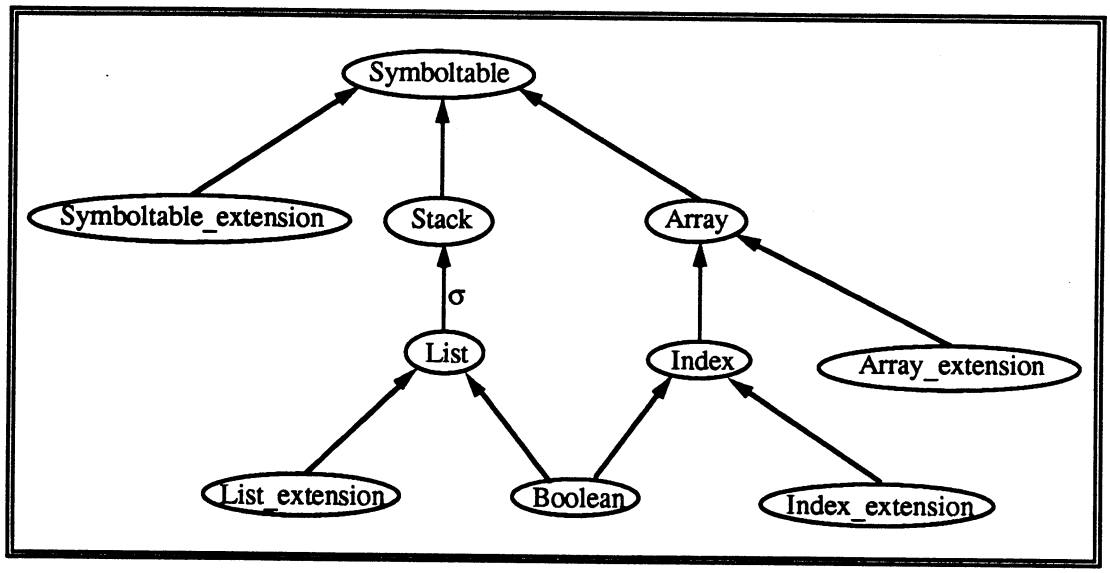
compilateurs de langage à structure de blocs. On utilise, entre autre, la théorie des piles et la théorie des tables[1] :

```
Array = extension of Index by :
types   array[t]
constants   nilarray : array[t]
            put : index x t x array[t] -> array[t]
            get : index x array[t] -> t
            isin : index x array[t] -> bool
axioms  get(i, put(i, x, a)) = x
        not(i=j) ==> get(i, put(j, x, a)) = get(i, a)
        isin(i, put(i, x, a)) = true
        not(i=j) ==> isin(i, put(j, x, a)) = isin(i, a)
        isin(i, nilarray) = false

Symboltable = extension of Stack and Array by :
constants   addid : index x t x stack[array[t]] -> stack[array[t]]
            retrieve : index x stack[array[t]] -> t
            isinblock : index x stack[array[t]] -> bool
            enterblock, leaveblock : stack[array[t]] -> stack[array[t]]
axioms  addid(i, x, st) = push(put(i, x, top(st)), pop(st))
        not(isin(i, a)) ==> retrieve(i, push(a, st)) = retrieve(i, st)
        isin(i, a) ==> retrieve(i, push(a, st)) = get(i, a)
        isinblock((i, st) = isin(i, top(st))
        enterblock(st) = push(nilarray, st)
        leaveblock(st) = pop(st)
```

Ce qui donne la structure plus complexe suivante :

[1]La théorie *Index* n'est pas explicitée ici, elle n'apporterait rien à la compréhension générale.



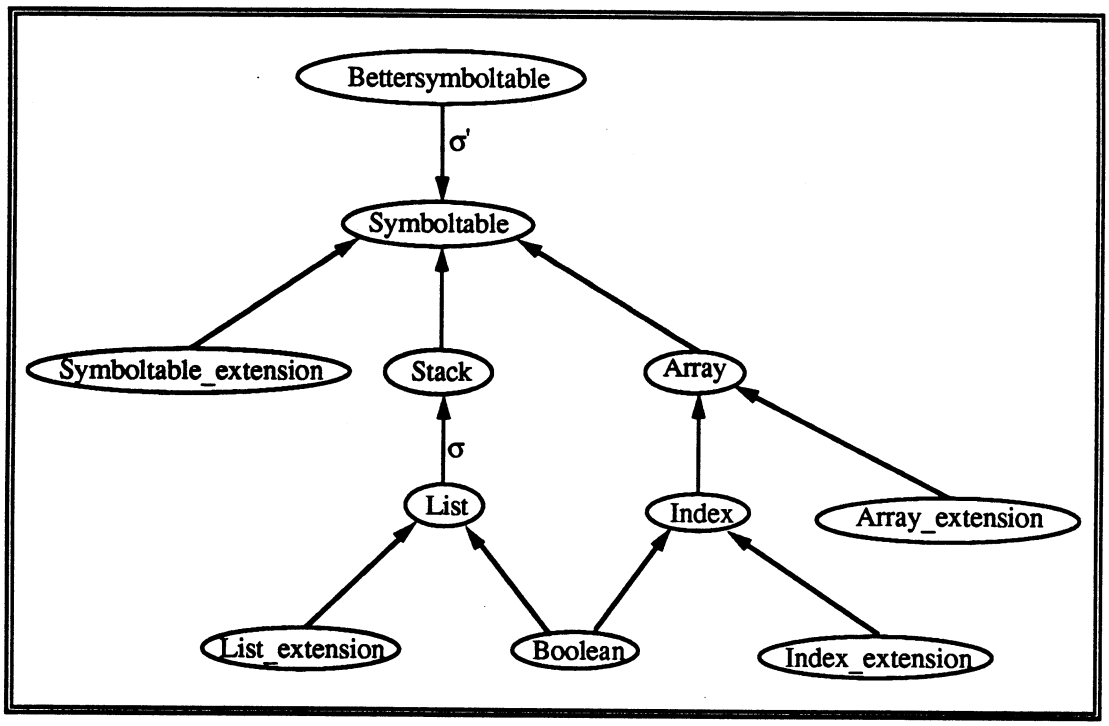
Théorie LCF des tables de symboles

En travaillant dans la théorie Symboltable, on est obligé d'utiliser le type *stack of array of t*. De même, on devra utiliser la constante *nilstack* plutôt que la constante plus naturelle *niltable*. De plus, la plupart des types, constantes et théorèmes des théories des piles ou des tables ne nous intéressent pas dans la théorie des tables de symboles. C'est dans le but d'éviter contraintes que l'on peut définir la notion d'image inverse qui consiste, en fait, à abstraire les définitions existantes et donc à redéfinir, via un renommage, ce qui nous intéresse.

Soient les nouvelle définitions :

```
types      symboltable[t]
constants  niltable : symboltable[t]
           addid : index x t x symboltable[t] -> symboltable[t]
           retrieve : index x symboltable[t] -> t
           isinblock : index x symboltable[t] -> bool
           enterblock, leaveblock : symboltable[t] -> symboltable[t]
```

On a alors la structure suivante :



Théorie LCF des tables de symboles avec image inverse

Dans cet exemple, le renommage est : σ'=({symboltable[t]→stack[array[t]]}, {niltable→nilstack, addid→addid, retrieve→retreive, isniblock→isinblock, enterblock→enterblock, leaveblock→leaveblock}). Les renommages des constantes et opérateurs des théories *Index* et *Boolean* sont considérés comme étant implicites et ne figurent donc pas dans l'énumération.

Dans une structure de cette forme, la méthode à suivre, afin de déterminer si une phrase de *Bettersymboltable* est un théorème, est de traduire cette phrase en utilisant le renommage σ' et de tenter de faire la démonstration dans la théorie des tables de symboles.

En plus de ces deux opérations de structuration (*rename* et *inv_image*), on peut ajouter l'opération de construction d'une théorie (*prim_theory*) qui prend une signature et des axiomes et retourne une théorie (construction des extensions, par exemple) et l'union (*union*) de deux théories (constructions des théories *List*, *Index*, etc...).

Grâce à ces différentes primitives, on peut écrire une théorie sous la forme d'une expression. Par exemple *rename(σ, union(prim_theory(Σlist, Slist), Boolean))* désigne la théorie structurée des piles (*Σlist* désigne la signature de la théorie *List*, *Slist* désigne les quatre axiomes de l'extension définissant la théorie *List* et σ est le renommage entre la théorie *List* et la théorie *Stack* précédemment définies).

On peut voir dans [SB 83] les définitions sémantiques précises de ces primitives de structuration de théories.

Avec cette façon de structurer les théories, il est nécessaire de mettre en œuvre certaines techniques pour réaliser des démonstrations. Pour répondre à la question *Est-ce que t est un théorème de la théorie rename(σ, Th)*, il faut regarder si il existe un théorème t' dans la théorie Th tel que t peut être obtenu à partir de t' grâce au renommage σ. Cette démarche peut nécessiter, bien sûr, la démonstration de t' dans Th.

Les différentes manières de structurer les théories LCF sont, en fait, spécifiées elle-même en LCF. On voit, sur l'exemple de *rename*, qu'une démonstration fait intervenir la sémantique de l'opérateur de structuration. Autrement dit, le fait même de spécifier les opérateurs de structuration va donner un moyen plus ou moins mécanique pour démontrer des théorèmes.

On peut voir dans [SB 83] les spécifications précises de ces opérateurs ainsi qu'un exemple de démonstration utilisant les spécifications.

6.3.3- Comparaison des différentes approches

Que ce soit en LPG ou en LCF, on manipule des spécifications sous forme de théories. Ces spécifications sont divisées en plusieurs modules reliés d'une manière plus ou moins compliquée, les uns aux autres. Cette démarche permet de manipuler des spécifications importantes en taille en conservant un certain contrôle.

Dans le cas de LCF traditionnel, les primitives de structuration sont simples (relation théorie mère $\leftrightarrow$ théorie fille). Il suffit alors de prendre les théorèmes précédemment démontrés ou les axiomes des théories ancestrales pour conduire une preuve.

Dans le cas de LCF enrichi de la notion de théories structurées, la sémantique des primitives de structurations est quelque peu plus compliquée. Cette idée tend surtout à simplifier la description des spécifications (*Il ne sert à rien de réinventer la roue...*). On définit des théories en utilisant plus intensivement celles qui existent déjà. Dans ce cadre, la spécification même des primitives de structuration indique la démarche à suivre pour conduire les preuves.

En ce qui concerne LPG, les spécifications sont également divisées en plusieurs modules, chaque module correspondant à une théorie. Les théories sont structurées les unes par rapport aux autres en utilisant des relations dont la sémantique est éventuellement plus riche que dans le cadre de LCF (ex : relation de satisfaction). Cette richesse est essentiellement accrue du fait que, non seulement on peut réutiliser facilement de précédentes définitions via d'éventuels renommages, mais la sémantique de certaines primitives de structuration impliquent la réalisation de divers contrôles. Ces contrôles prennent la forme de démonstration de théorèmes dans des théories bien précises. Autrement dit, la présence de certaines primitives de structuration indique que l'une des théories relative à la primitive doit contenir tel théorème pour être valide.

La démonstration de théorèmes est toutefois moins élégante que dans le système LCF. En effet, l'outil utilisé pour réaliser les démonstrations exige une

déstructuration des théories. Tous les axiomes qui pourront servir pour une démonstration sont vus au même niveau par le démonstrateur, quelle que soit la théorie à laquelle ils appartiennent effectivement.

On peut tout de même tirer profit de la structuration des théories de LPG. Grâce à cette structuration, on peut, en effet, ne donner au démonstrateur que ce qui pourra lui être utile pour réaliser une démonstration particulière et, par conséquent, limiter son champ d'exploration. Pratiquement, la rapidité des démonstrations en sera accrue.

6.4- Présentation étendue

Comme nous l'avons vu précédemment, afin de réaliser une démonstration dans un module LPG, il est nécessaire de réunir au même niveau tous les axiomes et théorèmes nécessaires déjà démontrés et de donner l'ensemble ainsi constitué au démonstrateur.

Plus formellement, la constitution de cet ensemble d'axiomes et de théorèmes peut être décrite par la notion de présentation étendue, de la manière suivante.

On notera $U \xrightarrow{d} U'$ pour indiquer que l'unité LPG U' dépend de l'unité U, U et U' étant présentée respectivement par P_U et $P_{U'}$. $\xrightarrow{d}$ désigne l'union des différentes relations pouvant exister entre des unités LPG, c'est à dire

- la relation d'importation $U_i \xrightarrow{i} U$,
- celle de paramétrisation $P \xrightarrow{r} U$,
- celle concernant la déclaration de modèle $P \xrightarrow{m} U$,
- la relation de satisfaction d'une propriété par une autre $P_s \xrightarrow{s} P$,
- la relation d'héritage $P_h \xrightarrow{h} P$ et enfin
- la relation de combinaison $\{P_c^{x,x=1,\dots,r}\} \xrightarrow{c} P$.

La présentation étendue $\tilde{P}_U$ de l'unité LPG U est l'ensemble des sortes, l'ensemble des opérateurs et l'ensemble des équations qui présente la théorie associée à U.

On peut définir récursivement $\tilde{P}_U$:

- si U est une unité indépendante[1], alors

  $\tilde{P}_U = P_U$,
- $\forall (U_j \xrightarrow{d_j} U')_{j=1,\dots k}$, on a

  $\tilde{P}_{U'} = \left(\cup F_{d_j}\left(\tilde{P}_{U_j}\right)\right)_{j=1,\dots,k} \cup P_{U'}$ où F_{d_j} est une fonction sur les présentations, définie par cas[2] :
  - $F_i(\Sigma,E)=(\Sigma,E)$ (importation)

[1]Une unité LPG est indépendante si elle n'utilise aucune autre définition que les siennes.

[2]$\Sigma^{\varnothing}$ désigne la signature vide $(S, \Omega)=(\varnothing, \varnothing)$.

- $F_r(\Sigma,E)=\begin{cases}\Phi^r(\Sigma,E) \text{ si U' n'est pas une propriété} \\ \text{non défini si U' est une propriété}\end{cases}$ (paramétrisation)
- $F_m(\Sigma,E)=\begin{cases}(\Sigma^{\varnothing},\varnothing) \text{ si U' n'est pas une propriété} \\ \text{non défini si U' est une propriété}\end{cases}$ (déclaration de modèle)
- $F_s(\Sigma,E)=(\Sigma^{\varnothing},\varnothing)$ (satisfaction)
- $F_h(\Sigma,E)=(\Sigma^{\varnothing},\Phi^h_E(E))$ (héritage)
- $F_c(\Sigma,E)=(\Sigma^{\varnothing},\Phi^c_E(E))$ (combinaison)

En d'autres termes, si une unité est indépendante, sa présentation étendue n'est autre que la présentation de l'unité elle-même.

Dans tous les autres cas, on considère la présentation de l'unité cible, *U'*, avec laquelle on fait l'union des présentations étendues de chacune des unités sources, U_j, après avoir réalisé d'éventuels renommages issus des morphismes définissant les relations.

Si l'unité cible U' est une propriété, les fonctions F_r et F_m ne sont pas définie puisqu'une propriété ne peut pas être la cible d'un morphisme de paramétrisation et qu'on ne peut pas y déclarer de modèle.

6.5- Les théorèmes

Les théorèmes que l'on est ammené à démontrer en LPG ont deux origines possibles. Soit ils font partie des déclarations d'une spécification. Ce sont alors des formules qui doivent être des conséquences logiques de la spécification. Soit ils sont générés automatiquement afin que leur démonstration puisse valider certaines spécifications. Nous ne nous intéresserons, dans cette partie, qu'à cette deuxième catégorie de théorèmes.

Ces théorèmes sont issus des propriétés sources des morphismes de déclaration de modèles, Φ^m, et des morphismes de satisfaction entre propriétés, Φ^s. Soit une équation e appartenant à une propriété P telle que $P \xrightarrow{m} U$ (resp. $P \xrightarrow{s} U$), la formule obtenue à démontrer est $\Phi^m_E(e)$ (resp. $\Phi^s_E(e)$).

Etant donnée une propriété *P*, source d'un morphisme de déclaration de modèle ou de satisfaction, les équations qui sont à l'origine des formules à démontrer appartiennent à la théorie étendue associée à *P*.

Cet ensemble d'équations, $\overset{*}{E}_P$, sera constitué des équations de la présentation $\overset{*}{P}_P=(\overset{*}{\Sigma}_P,\overset{*}{E}_P)$ que l'on peut définir récursivement de la manière suivante :

- si P est une propriété indépendante présentée par $P_P=(\Sigma,E)$, on a $\overset{*}{P}_P=P_P$,
- $\forall(P_j \xrightarrow{d_j} P')_{j=1,...,k}$ et si P' est présentée par $P_{P'}=(\Sigma',E')$, alors

$\overset{*}{P}_{P'}=\left(\cup F'_{d_j}\left(\overset{*}{P}_{P_j}\right)\right)_{j=1,\ldots,k}\cup P_{P'}$ où $F'_{d_j}(\Sigma, E)$ est une fonction sur les présentations définie par cas[1] :

- $F'_i(\Sigma,E)=(\Sigma^{\varnothing},\varnothing)$ (importation)
- $F'_r(\Sigma,E)$: non défini (paramétrisation)
- $F'_m(\Sigma,E)$: non défini (déclaration de modèle)
- $F'_s(\Sigma,E)=(\Sigma^{\varnothing},\varnothing)$ (satisfaction)
- $F'_h(\Sigma,E)=(\Sigma^{\varnothing},\Phi^h_E(E))$ (héritage)
- $F'_c(\Sigma,E)=(\Sigma^{\varnothing},\Phi^c_E(E))$ (combinaison)

Ce qui revient à dire que si on a affaire à une propriété indépendante, l'ensemble des équations est constitué des équations figurant dans sa présentation.

Dans les autres cas, on réalise par fermeture transitive l'union entre l'ensemble des équations de la propriété cible des relations et ceux des propriétés sources, après avoir été renommées par les morphismes définissant les relations.

Les relations considérées ici sont les relations d'héritage et de combinaison. Par ailleurs, F'_r n'est pas définie puisqu'une propriété ne peut pas être générique. F'_m n'est pas définie non plus car on ne peut pas déclarer de modèle dans une propriété.

6.6- Le logiciel

6.6.1- Les possibilités d'interfaçage

Concrètement, le problème est alors de réunir les logiciels existant LPG et OASIS de manière à utiliser les fonctionnalités de démonstration de OASIS alors que l'utilisateur travaille avec son environnement sous le contrôle de LPG. On appellera à partir de maintenant le logiciel réalisant ce lien *interface*. Autrement dit, on aimerait utiliser les différentes fonctions de démonstrations offertes par OASIS sur les définitions qui sont écrites dans LPG. Il faudrait donc que OASIS puisse manipuler d'une certaine manière les spécifications écrites en LPG.

Le tout premier problème à résoudre concerne le choix du langage dans lequel sera écrit l'interface. Il faut savoir tout d'abord que LPG et OASIS fonctionnent tous les deux sur le système d'exploitation Multics, mais que leurs langages d'implémentation sont différents. LPG est écrit en PL1. OASIS, quant à lui, est écrit en PROLOG. Bien évident, les structures de données manipulées pour représenter l'information utile, en fait les équations, sont totalement différentes.

Il est donc nécessaire que le langage choisi soit standard, puisse accéder aux structures de données manipulées par LPG, aux structures de données manipulées par OASIS (à moins de passer par l'intermédiaire de fichiers, ce qui est à éviter

[1] $\Sigma^{\varnothing}$ désigne la signature vide $(S,\Omega)=(\varnothing, \varnothing)$.

pour des raisons de performances et de convivialité) et puisse activer les fonctionnalités de OASIS.

Le langage choisi pour réaliser cette interface a été PASCAL. Il répond en effet aux différents critères de sélection vus précédemment.

Le choix du langage, les problèmes de partage de structure de données et les problèmes d'appel des fonctions du démonstrateur étant résolus, il est nécessaire de trouver un moyen pour transmettre l'information (les équations) de LPG à OASIS.

En ce qui concerne LPG, une équation est représentée comme une arborescence dont la racine est le symbole de l'égalité entre deux termes. Chaque terme est lui-même représenté par une arborescence. Ces arborescences sont représentées par différentes tables PL1.

OASIS, quant à lui, manipule des règles de réécriture qui sont représentées comme des faits PROLOG.

Il faut donc de transformer les équations de LPG en parcourant les diverses tables en des clauses PROLOG. Voyons les diverses solutions envisageables.

Une première solution pourrait être une traduction des spécifications LPG en spécifications OASIS. Il suffirait alors d'appeler OASIS et de faire les démonstrations après avoir pris en compte les nouvelles spécifications. Cette solution très simple utiliserait des fichiers pour enregistrer les traductions de spécifications que pourrait manipuler OASIS.

Comme on l'a vu précédemment, l'utilisation de fichiers peut s'avérer pénalisante en ce qui concerne le temps d'exécution. L'autre problème est la convivialité. Il faut changer d'environnement de travail pour réaliser des démonstrations. On demande à LPG de faire une traduction de spécifications. On quitte le système LPG pour appeler le démonstrateur OASIS. On donne en entrée de ce dernier les traductions et on réalise les démonstrations. Le travail terminé, il faut quitter OASIS pour revenir à l'environnement LPG et prendre en compte le résultat des démonstrations. Cette solution, bien que simple, n'est pas satisfaisante.

Les deux problèmes essentiels qui se posent sont d'une part le moyen pour transférer l'information et d'autre part comment appeler les fonctionnalités du démonstrateur depuis le système LPG.

Ces deux problèmes étant indépendants, on peut esssayer de les résoudre séparément.

Supposant le problème d'appel des fonctionnalités résolu, on peut imaginer comment faire le transfert de l'information. Le passage par fichier source (traduction simple des spécifications en OASIS) étant éliminé, il serait possible de créer directement à partir de LPG les fichiers binaires utilisés par OASIS, on éviterait ainsi une étape de compilation intermédiaire OASIS. On peut voir le format interne de ces fichiers dans les annexes. Cette solution n'est toutefois toujours pas vraiment intéressante du fait qu'elle fait intervenir des fichiers et donc que les performances en seront dégradées.

Le langage de l'interface permet d'accéder directement aux structures de données internes des deux logiciels. Une autre solution consiste à générer directement dans les structures de données de OASIS les définitions dont ce dernier aura besoin pour réaliser les démonstrations. Il suffit pour cela de connaître précisément les structures manipulées, ce qui peut ne pas être simple. Cette solution présente par contre l'avantage de ne pas perdre de temps en manipulation de fichier.

Concernant l'appel des fonctionnalités de démonstration, après quelques investigations, il s'est avéré qu'un point d'entrée particulier, appelé *sprolog*, de l'interpréteur PROLOG permet d'évaluer un prédicat passé en paramètre. Par ailleurs, il est possible de demander à PROLOG d'évaluer des prédicats qui lui sont fournis non pas sur le support d'entrée traditionnel (en l'occurence, un terminal), mais à partir de buffers internes bien précis. L'interpréteur PROLOG étant lui-même écrit en PASCAL, ces buffers internes ne sont en fait rien d'autre que des tableaux PASCAL qui peuvent aisément se partager avec des déclarations réalisées dans l'interface.

Chaque fonctionnalité du démonstrateur est implémentée par un paquet de clauses PROLOG. Quand on fait appel à une fonctionnalité, on demande à évaluer le prédicat correspondant avec des arguments précis.

La solution retenue concernant l'appel de ces fonctionnalités consiste à mettre dans l'un des buffers de PROLOG le prédicat à évaluer et à faire appel au point d'entrée précédemment signalé en disant simplement que la suite des évaluations se réalisera à partir du buffer utilisé. Après quoi, sous le contrôle du système LPG, on place les commandes du démonstrateur que l'on désire activer dans ce buffer interne. Le système OASIS étant averti de cette configuration prend en compte le contenu du buffer, exécute la commande[1] et délivre le résultat dans un autre buffer qui est lu par le système OASIS.

Après avoir mis au point cette solution, le transfert des informations entre LPG et OASIS devient relativement simple. Nous avons vu que les équations manipulées par OASIS sont représentées par des faits PROLOG. Il suffit alors de récupérer une équation provenant de LPG et de demander à PROLOG, grâce au point d'entrée *sprolog* et aux buffers particuliers, d'enregistrer la traduction de cette équation comme un nouveau fait pour la base de donnée PROLOG. Réciproquement, les résultats calculés par le démonstrateur (pratiquement, tel théorème vient d'être démontré) pourront être transmis à l'interface via ces mêmes buffers.

Le problème qui reste à résoudre est alors de réaliser une traduction systématique des équations LPG en règle de réécriture OASIS. Accessoirement, il faudra également se plier à quelques exigences de OASIS et enregistrer quelques définitions supplémentaires nécessaires au bon fonctionnement du démonstrateur, comme par exemple une liste des noms, avec leur profile, des opérateurs définis.

---

[1]Ce qui exige généralement une interaction avec l'utilisateur.

6.6.2- Principe

Le principe adopté pour réaliser l'interface entre LPG et OASIS est donc le suivant.

L'interface, qui va réaliser le lien entre les deux logiciels, se chargera d'appeler le système OASIS en demandant d'évaluer des clauses. Ces clauses, d'une part activeront les fonctionnalités du démonstrateur, et d'autre part permettront d'enregistrer l'information manipulable par OASIS et, en retour, de rendre compte des démonstrations réalisées.

Cette connexion offre deux nouvelles fonctionnalités pour LPG. La plus simple consiste à réaliser des démonstrations de théorèmes écrits dans les spécifications de LPG. La deuxième fonctionnalité, qui est en fait la raison d'être de l'interface, est de valider sémantiquement des spécifications écrites en LPG en utilisant les résultats vus dans le cinquième chapitre de cette thèse (relations sémantiques). Cette validation revient en fait à vérifier que des formules générées automatiquement à partir des spécifications sont des théorèmes dans dans théories précises.

Schématiquement, l'utilisateur entamera une session de travail sous LPG en écrivant des spécifications. Au fur et à mesure de sa progression, il pourra demander à valider ce qu'il vient d'écrire ainsi que démontrer des formules qui doivent être des conséquences logiques dans la théorie qu'il est en train de construire. Il faut noter qu'en aucun cas, il ne sera obligé quitter le contrôle de LPG pour réaliser ses démonstrations. Les appels au démonstrateur sont réalisés de manière transparente.

Les demandes de résolution de clauses communiquées au système OASIS (ou, plus exactement à PROLOG) se présentent sous la forme

-EXTERNALIO -lpg(<commande>) -stop.

-EXTERNALIO indique à l'interpréteur qu'il doit prendre en compte les commandes se trouvant dans un buffer particulier.

Vient ensuite la demande de résolution de *-lpg(<commande>)*. La clause *-lpg* permet de vérifier que la commande demandée existe et est autorisée. *<commande>* est le nom de la commande que l'on désire utiliser, préfixé par la chaîne de caractères *lpg* afin d'éviter d'éventuels conflits de nom, suffixé éventuellement de paramètres[1] et terminé par la chaîne de caractères *."0"*[2].

-stop permet de rendre le contrôle à l'interface.

Prenons, par exemple, la commande *cas_automatique 1 6* qui demande de faire une analyse automatique du nœud repéré par *1 6*. La clause communiquée à PROLOG est

---

[1]Chaque paramètre est ajouté au nom en le faisant précédé d'un point.

[2]Le nom de la commande avec ses éventuels paramètres constituent une liste. Pour l'interpréteur PROLOG, la fin de liste est représentée par cette chaîne de caractères *."0"*.

-EXTERNALIO -lpg(lpgcas_automatique.1.6."0") -stop..

6.6.3- Restrictions

Il n'est actuellement pas possible de valider sémantiquement tout ce que l'on peut écrire en LPG. Divers points ne sont en effet pas encore pris en compte. Le but recherché était avant tout de réaliser une maquette qui prenne en compte les caractéristiques générales du langage (utilisation des fonctions, de la généricité, des instanciations, etc). La prise en compte des points particuliers qui ne sont pas encore traités ne devrait généralement pas poser de problème.

Il faut remarquer qu'il y a deux niveaux de prise en compte des caractéristiques du langage. Une caractéristique est une particularité du langage. Par exemple, à la caractéristique (notion abstraite de) fonction correspond la déclaration syntaxique des fonctions, leur définition sémantique et leur utilisation.

Le premier niveau de prise en compte d'une caractéristique consiste simplement à explorer une notation afin de détecter d'éventuels liens existant entre l'unité que l'on est en train de traiter et d'autres unités LPG. Quand on rencontre, par exemple, une équation définissant un opérateur, ce premier niveau de prise en compte ne fait que regarder les deux membres de l'équation afin de détecter les nouveaux opérateurs utilisés.

Le deuxième niveau consiste à traiter effectivement la notation rencontrée. Par exemple, face à une équation définissant un opérateur, il faut traduire cette équation dans le formalisme de OASIS.

Les restrictions mentionnées concerne essentiellement le deuxième niveau de prise en compte. Selon les cas, le premier niveau peut ou non être traité.

Voici les divers points qui ne sont pas encore considérés par l'interface.

• Les constantes de type produit cartésien ainsi que la projection du produit cartésien ne sont pas traités. Le type produit cartésien n'est pas un type prédéfini de OASIS. La traduction n'est donc pas immédiate et devrait passer vraisemblablement par la spécification explicite en OASIS d'un type abstrait *Produit_Cartésien*. Le type produit cartésien n'étant pas traité, il en est de même pour l'opérateur de projection. Toutefois, quand on se trouve en présence d'une constante de type produit cartésien ou d'une projection, l'expression est analysée afin de détecter les liens éventuels avec d'autres unités (premier niveau de traitement).

• L'opérateur de curryfication n'est traité qu'au premier niveau. Le deuxième niveau nécessite une étude particulière car cette notion n'existe pas en OASIS. Cette étude n'a pas été réalisée, l'intérêt apporté par ce traitement semblant ne pas justifié le temps nécessaire à sa réalisation.

• Les prédicats ainsi que les clauses les définissant ne sont pris en compte à aucun niveau. En effet, au moment de la réalisation du logiciel de l'interface, ce concept n'existait pas encore dans le langage. Leur traitement ne semble pas, par contre, poser de très grosses difficultés. Une étude assez rapide devrait suffire pour les inclure dans l'interface.

• De même que pour les prédicats, les exceptions ne sont pas du tout traitées. Leur sémantique est relativement différente dans les deux systèmes. En OASIS, les exceptions sont implémentées par des équations particulières appelées préconditions. Avant chaque appel à un opérateur *op*, le système regarde s'il existe une ou plusieurs préconditions associées. Si c'est le cas, l'évaluation de *op* n'est réalisée que si toutes les préconditions sont évaluées à *VRAI*. A la première précondition évaluée à *FAUX*, l'évaluation toute entière (non seulement celle de *op*, mais également celle de l'opérateur éventuel qui avait appelé *op*) s'arrête en émettant un message d'erreur comme résultat général de l'évaluation. Un cas d'exception n'est donc pas rattrapé. En LPG une exception peut être déclenchée lors de l'évaluation d'une expression. De même que pour OASIS, on peut spécifier des cas exceptionnels pour les opérateurs partiellement définis par des équations particulières également appelées préconditions. Un tel opérateur ne sera évalué que si sa précondition est vérifiée. Par contre, il est possible de récupérer une exception. C'est à dire que l'évaluation globale ne sera pas stoppée à cause de l'exception rencontrée, mais continuera avec une valeur précise calculée par le récupérateur de l'exception. Il faudrait donc étudier ce problème en détail pour prendre en compte les exceptions.

6.6.4- Création de l'ensemble des théorèmes

Etant donnée la description formelle (voir la section cinq de ce chapitre) de la constitution de l'ensemble des théorèmes qu'il faut démontrer pour valider une certaine unité LPG *U*, voici présentée plus informellement la créationde cet ensemble de théorèmes.

Les théorèmes que l'on aura à démontrer sont générés à partir des déclarations de modèles et des morphismes de satisfaction.

Soit la déclaration de modèle[1]

$$P[S_1, ..., S_m \ operators \ F_1, ..., F_n]$$

réalisée dans l'unité LPG *U*. L'ensemble des théorèmes générés par cette déclaration sera constitué d'une part des équations de *P* après avoir réalisé la substitution correspondant au morphisme et d'autre part des équations (toujours obtenues après avoir réalisé la substitution correspondant aux différents morphismes correspondants) provenant des propriétés héritées ou combinées par *P*, ainsi que des équations issues de l'application récursive de ce traitement sur chacune des propriétés héritées ou combinées.

En d'autres termes, on récupère les équations de *P* en les renommant et on applique récursivement ce traitement sur chaque propriété héritée ou combinée de *P*.

L'ensemble des théorèmes à démontrer issus de la déclaration d'un morphisme de satisfaction sera construit sur le même principe.

---

[1]Dans ce chapitre, nous omettrons l'utilisation des prédicats puisqu'ils ne sont pas pris en compte dans le logiciel existant.

6.6.5- Construction de la présentation étendue

La construction de la présentation étendue a été décrite formellement dans la section quatre de ce chapitre. Pratiquement, un ensemble d'équations est créé. Cet ensemble est construit de la manière suivante.

On initialise tout d'abord cet ensemble avec les équations se trouvant dans l'unité LPG dans laquelle est déclaré le morphisme à valider. On complète ensuite cet ensemble par les équations provenant des unités transitivement importées à condition toutefois qu'une telle unité ait été déjà validée sémantiquement. On complète alors l'ensemble obtenu par les équations provenant de l'éventuelle propriété exigée (ainsi que celles issues récursivement des propriétés qu'elle hérite ou combine ou des autres unités qu'elle importe) en effectuant chaque fois le renommage adéquat et à condition que chaque unité ait été déjà validée.

Comme on l'a vu précédemment, la structuration des théories permet ainsi de limiter le nombre d'axiomes à fournir au démonstrateur et donc d'accroitre les performances.

On peut remarquer que l'importation $U_s \xrightarrow{i} U_t$, telle qu'elle a été décrite précédemment réalise l'inclusion de la totalité des définitions de U_s dans U_t. Or il est possible que certaines définitions de U_s ne soient pas utiles pour réaliser les différentes démonstrations.

Tenant compte de cette remarque, ce n'est pas l'importation telle qu'elle a été présentée qui a été implémentée, mais plutôt une restriction. L'ensemble des équations ainsi généré fait partie de ce qu'on appellera un *environnement de démonstration.*

On réalise ainsi simplement une exploration de chaque équation qui entre dans cet environnement de démonstration et, pour chaque utilisation d'un nouvel opérateur, on récupère les équations le définissant et uniquement celles-ci.

En effet, le démonstrateur fonctionnant essentiellemnt par réécriture, il n'est pas nécessaire d'avoir des équations définissant des opérateurs inutilisés par les théorèmes à démontrer ou les différentes équations qui en découlent. Cette remarque permet de diminuer encore le nombre d'axiomes à manipuler par le démonstrateur.

L'environnement de démonstration contient, en plus des équations ainsi collectées, des informations nécessaires pour le système OASIS (ex : profile des opérateurs utilisés) et les théorèmes qu'il faut essayer de démontrer.

6.6.6- Structure du logiciel

6.6.6.1- Principe

Afin de réaliser les connexions entre l'interface et LPG et OASIS, il a été nécessaire de modifier quelque peu les systèmes LPG et OASIS.

Les modifications de LPG sont assez superficielles et se divisent en deux catégories. D'une part il a fallu faire en sorte que l'on puisse se servir des fonctionnalités de démonstration de l'interface sous le contrôle du système LPG.

Cette modification est évidente et concerne le module d'interprétation des commandes de LPG. Il suffit en effet de lui rajouter la reconnaissance des commandes nécessaires. D'autre part, l'interface a besoin de diverses fonctionnalités que LPG doit lui offrir, comme par exemple le nombre d'arguments d'un opérateur. La plupart de ces fonctionnalités existaient déjà, mais il a fallu tout de même en définir certaines autres.

Celles de OASIS, par contre, sont plus importantes. OASIS est écrit en PROLOG et quand on appelle une fonctionnalité particulière du démonstrateur, c'est en fait la résolution d'un prédicat qui est demandée.

Le lancement de OASIS se traduit simplement par l'initialisation d'une session PROLOG en utilisant un environnement particulier, qui contient des définitions propres à OASIS. Cet environnement peut être généré une bonne fois pour toutes en procédant à une compilation des définitions et en sauvegardant l'environnement ainsi constitué. Cet environnement, dans la terminologie de l'interpréteur PROLOG, s'appelle un *prélude*.

On a vu que le principe de communication entre l'interface et OASIS consistait à demander la résolution d'une clause se trouvant dans un buffer spécial de PROLOG. Il est donc nécessaire qu'il existe des définitions de prédicats correspondant aux différentes demandes de résolution possibles. Ces définitions seront données dans un fichier particulier appelé *interf.prolog*.

Les prédicats doivent être connus du système OASIS, ce qui fait qu'il a été nécessaire de générer un nouvel environnement pour le fonctionnement de OASIS en prenant en compte le nouveau fichier *interf.prolog*. On établit la liste des différentes fonctionnalités que l'on désire avoir. Il suffit alors de les programmer sous formes de clauses PROLOG que l'on rajoute à l'environnement initial définissant OASIS.

Une autre modification de OASIS a été nécessaire. Elle concerne la forme des identificateurs admis par OASIS. Cette modification étant essentiellement technique, elle figure en annexe de cette thèse.

6.6.6.2- Génération de OASIS

Nous allons expliquer très brièvement comment se passe la génération de l'environnement de OASIS.

On commence par appeler l'interpréteur PROLOG en lui demandant de charger un certain nombre de définitions standards. Parmis ces définitions standards, on trouve des définitions assurant la compatibilité avec des versions antérieurs de l'interpréteur, des définitions permettant de s'adapter aux particularités du système d'exploitation, etc.

On charge alors les clauses définissant le système OASIS lui-même. On peut ensuite créer un prélude. Ce dernier peut être binaire (en fait, le résultat d'une espèce de compilation des différentes clauses contenues dans la mémoire PROLOG) ou se présenter sous forme de texte PROLOG. La seule différence entre les deux possibilités est le temps de chargement pour une utilisation ultérieure du prélude. Ce temps est plus court pour la forme codée que pour la forme texte.

Le lancement de OASIS, quant à lui, se ramène au lancement de PROLOG avec ce prélude comme ensemble de définitions initiales. On peut également demander à PROLOG de fonctionner, non pas avec ce prélude initial, mais avec un prélude tel qu'il a été laissé lors de la dernière session d'utilisation de OASIS[1]. Cela permet de récupérer des définitions qui ont été précédemment élaborées.

En effet, le prélude initial, après avoir été chargé, constitue un environnement contrenant diverses définitions. Cet environnement est généralement complété au fur et à mesure de son utilisation par adjonction de nouvelles définitions comme, par exemple, la définition de nouveau types.

6.6.6.3- L'interface

Le logiciel de l'interface est composé de cinq modules différents écrits dans des langages de programmation éventuellement différents du fait de l'hétérogénéité des logiciels à relier. Trois de ces modules réalisent effectivement l'implémentation de l'interface, les deux autres permettent de relier l'interface d'une part au système LPG et d'autre part au système OASIS. Un sixième fichier, constitué de clauses PROLOG, a du être écrit afin de pouvoir générer un nouvel environnement de fonctionnement pour OASIS.

La figure suivante décrit la structure générale de l'interface. On y voit à gauche LPG, la partie de droite concerne OASIS et entre les deux, les cinq modules de l'interface. En dessous figurent également les buffers utilisés pour transmettre l'information entre l'interface et OASIS. Les flèches noires (⟶) indiquent l'appel de la part d'un module à une fonction d'un autre module, les flèches grises (⇢) traduisent un transfert d'information entre différents modules.

La partie concernant LPG (*lpg.pl1*) n'a pas été détaillée du fait de sa complexité. De plus, les modifications concernent essentiellement le module *lpg.pl1* présenté.

Le système OASIS, quant à lui, est constitué de l'interpréteur PROLOG (schématiquement appelé *prolog.pascal*). On trouve également un fichier appelé *predext.pascal*, qui contient différentes définitions de prédicats cablés, comme par exemple un prédicat d'appel à une commande du système d'exploitation, ainsi que la procédure qui réalise la lecture des identificateurs. On voit enfin les deux types de préludes, binaire (*preludebin.oasis*) ou source (*prelude.oasis*), qui contiennent les définitions du système OASIS lui-même ainsi que quelques fichiers permettant d'initialiser une session OASIS (*prelude.binaire, prelude.reprise*).

On peut voir les deux modules réalisant les liens entre l'interface et les deux systèmes :

- *adapt.pl1* établit le lien avec le système LPG,
- *interf.pl1* réalise le lien avec le système OASIS.

Ces deux modules ont été écrits en PL1.

---

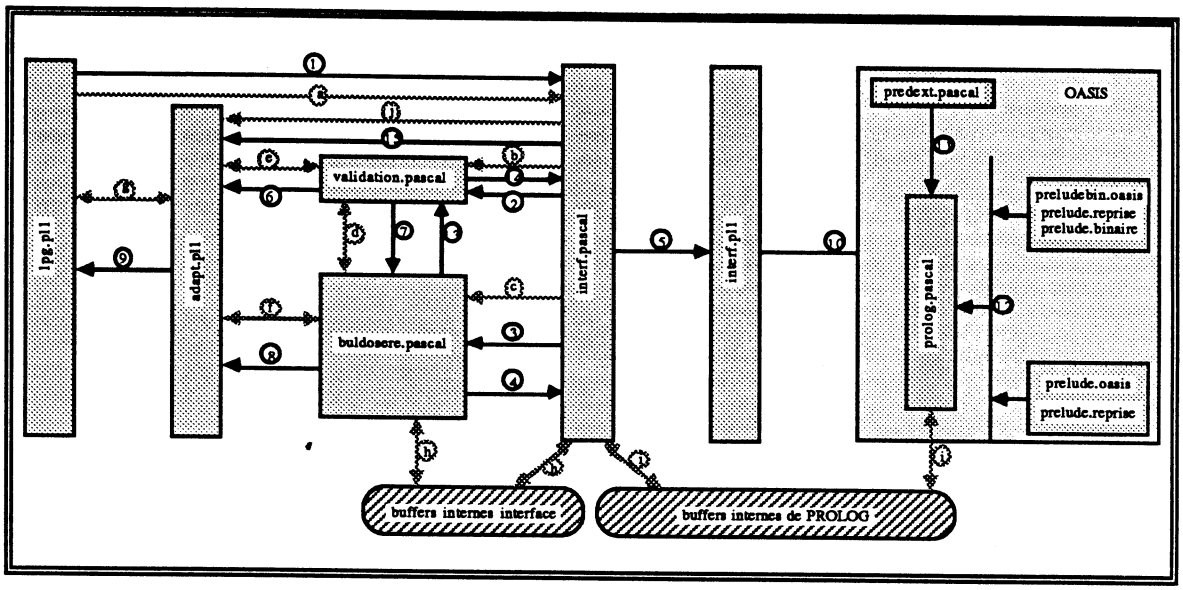
[1] A condition, toutefois, que cette dernière session ait été réalisée durant la même session de connexion au calculateur.

Enfin, on trouve les trois modules propres à l'interface :

• *interf.pascal* qui offre les différents points d'entrée de l'interface à LPG et assure la coordination générale de l'interface,

• *validation.pascal* qui s'occupe de l'aspect validation de l'interface comme, par exemple, déterminer quelles sont les différentes unités LPG à valider pour qu'une certaine unité initiale soit elle-même valide et

• *buldosere.pascal* qui se charge d'aplatir les théories structurées qu'on lui fournit en paramètre et qui crée l'environnement de démonstration pour le système OASIS.



Structure générale de l'interface

Expliquons maintenant l'utilité de chacune des flèches de ce schéma. Les points préfixés par un numéro décrivent des appels entre différents modules. Les points préfixés par un caractère décrivent les transferts d'informations.

1- Le système LPG peut appeler l'interface par deux points d'entrée :

• prouver : permet de tenter de démontrer les théorèmes se déclarés dans une unité,

• valider : permet de tenter de valider une unité par démonstration successive de théorèmes.

2- L'interface peut appeler le module de validation par trois points d'entrée :

• listavalider(u) : fonction qui retourne la liste des différentes unités LPG qu'il faut valider pour que l'unite *u* fournie en paramètre soit valide,

• validerautomatique(u) : tente de valider, de manière automatique (c'est à dire sans intervention de l'utilisateur) l'unité *u* passée en paramètre,

• valider(u) : initialise une session de validation concernant l'unité *u* fournie en paramètre.

3- L'interface peut appeler le module de preuve par un point d'entrée :

• prouvertheo(u) : initialise une session de démonstration concernant les théorèmes déclarés dans l'unité *u* donnée en paramètre.

4- le module de preuve peut appeler l'interface par un point d'entrée :

• decharg1clauseoasis : transfert une clause pour OASIS du buffer interne de l'interface vers le buffer interne de PROLOG et demande, ensuite, la résolution de cette clause à PROLOG.

5- L'interface peut appeler le module intermédiaire à PROLOG par deux points d'entrée :

• initoasis : initialise l'environnement de OASIS,

• oasis : demande l'utilisation du prélude de la session précédente et demande la résolution du prédicat se trouvant dans le buffer interne de PROLOG.

6- Le module de validation peut appeler le module intermédiaire à LPG par toute une série de points d'entrée correspondant chacun à une manipulation des tables internes à LPG qui contiennent les définitions des unités LPG.

7- Le module de validation peut appeler le module de preuve par trois points d'entrée :

• prouvermorphisme(m) : complète l'environnement de démonstration correspondant au morphisme *m* et initialise une session de démonstration OASIS,

• prouvermodele(m) : complète l'environnement de démonstration correspondant au modèle *m* et initialise une session de démonstration OASIS,

• creertheomorphisme(m) : réalise la création des théorèmes relatifs au morphisme *m* à valider,

• creertheomodele(m) : réalise la création des théorèmes relatifs au modèle *m* à valider.

8- Le module de preuve peut appeler le module intermédiaire à LPG par toute une série de points d'entrée correspondant aux fonctions de manipulation des tables internes à LPG.

9- Le module intermédiaire à LPG peut appeler les différents points d'entrée offerts par le système LPG. Ce module permet de centraliser les points d'entrée de LPG nécessaire au fonctionnement de l'interface.

10- Le module intermédiaire à PROLOG peut appeler le système OASIS par un point d'entrée :

• sprolog(string) : réalise un appel à l'interpréteur PROLOG en demandant de résoudre la clause représentée par la chaîne de caractère *string* fournie en paramètre.

11- Le programme *predext.pascal* permet de définir des prédicats câblés fournis par l'utilisateur.

12- Un prélude, nécessaire pour le bon fonctionnement de PROLOG, doit exister. Il permet de définir les prédicats de base. Il peut exister sous deux formes :

• forme source : on y trouve des paquets de clauses PROLOG,

• forme objet : représentation compilée de la forme source. Il est nécessaire doit lui adjoindre un prélude source spécial (*prelude.binaire*).

Dans les deux cas, un autre prélude spécial est nécessaire pour pouvoir reprendre une session interrompue (*prelude.reprise*).

13- Le module de preuve peut appeler le module de validation par trois points d'entrée :

• validerautomatique(u) : essaie de valider de manière automatique l'unité LPG *u*,

• validerautomatiquemorphisme(m) : essaie de valider de manière automatique le morphisme *m*,

• validerautomatiquemodele(m) : essaie de valider de manière automatique le modèle *m*.

14- Le module de validation appelle l'interface par deux points d'entrées :

• prouvermorphisme(m) : demande de réaliser la validation du morphisme *m*,

• prouvermodèle(m) : demande de réaliser la preuve du modèle *m*.

15- L'interface peut appeler le module intermédiaire à LPG par différents points d'entrée afin de manipuler les tables internes à LPG.

a- Le système LPG peut transmettre à l'interface une chaîne de caractères représentant l'identificateur d'un unité LPG.

b- L'interface transmet au module de validation le numéro de l'unité de travail.

c- L'interface transmet au module de preuve le numéro de l'unité dont il faut prouver les théorèmes.

d- Le module de validation transmet au module de preuve le numéro de l'entité sur laquelle le démonstrateur aura à travailler.

e- Le module de validation transmet (récupère) diverses informations au (du) module intermédiaire à LPG.

f- De la même manière, le module de preuve transmet (récupère) diverses informations au (du) module intermédiaire à LPG.

g- Le module intermédiaire assure le transfert (la récupération) de ces informations vers le (provenant du) système LPG.

h- Le buffer interne à l'interface permet l'échange d'information (clauses construites par le module de preuve) entre le module de preuve et le module général de l'interface.

i- Les buffers internes à PROLOG permettent de transférer les informations (clauses PROLOG) entre l'interface et le système PROLOG. Le module général de l'interface peut éventuellement transformer les clauses provenant du module de preuve afin de les adapter pour PROLOG.

j- L'interface transmet (récupère) diverses informations vers le (du) module intermédiaire à LPG.

6.6.7- Les commandes

6.6.7.1- Introduction

On présente dans cette partie les différentes commandes mises à la disposition de l'utilisateur par l'interface. On peut ainsi avoir une vue d'ensemble des possibilités offertes par l'interface.

Au cours d'une session de démonstration, la preuve est représentée par une arborescence que l'utilisateur peut manipuler. Les feuilles de cette dernière sont, à un instant donné, les sous-buts à atteindre. Par ailleurs, le système d'inférence (règles de réécriture, schémas d'induction) ainsi que les théorèmes et diverses autres informations sont représentés par des clauses PROLOG. L'ensemble de ces clauses a été appelé environnement de démonstration.

On va ainsi trouver une catégorie de commandes permettant la manipulation de l'arborescence de démonstration et une deuxième catégorie concernant la gestion de l'environnement de démonstration. Une troisième catégorie de commande concerne différentes aides en ligne relatives aux techniques de démonstration employées dans notre cadre. Enfin, une quatrième catégorie regroupe diverses commandes permettant, par exemple, de modifier les stratégies employées pour la réécriture.

Il faut remarquer que ces commandes ne font, pour la plupart, qu'appeler des commandes existantes de OASIS. Cette caractéristique sera signalée afin de pouvoir éventuellement se référer à la partie du chapitre trois de cette thèse qui les décrit.

6.6.7.2- Gestion de l'arborescence

Ces commandes s'occupent de la construction et éventuellement de la destruction de l'arborescence de démonstration.

cas_automatique <pos> : génération automatique de sous-buts à partir du but présent au nœud <pos>. Cette commande a déjà été décrite.

cas_manuel <pos> : génération de sous-buts à partir du but présent au nœud *<pos>* suivant une décomposition fournie par l'utilisateur. Cette commande a déjà été décrite.

élaguer <pos> : permet de recommencer une démonstration partiellement ou en totalité. Cette commande a déjà été décrite.

induction <pos> : activation d'un processus d'induction. Cette commande a déjà été décrite.

prouver <nom> <entier> : permet d'initialiser une nouvelle session de preuve. Cette commande a déjà été décrite.

reducas <pos> : réduction par réécriture de tous les nœuds fils du nœud *<pos>*. Cette commande a déjà été décrite.

reduction <pos> : réduction par réécriture du nœud *<pos>*. Cette commande a déjà été décrite.

6.6.7.3- Visualisation et déplacement dans l'arborescence

Ces commandes permettent de visualiser l'arborescence de démonstration ainsi que changer les nœud de travail courant.

cas_courant : visualisation du sous-but courant. Cette commande a déjà été décrite.

cas_restant : donne les sous-buts considérés comme non démontrés. Cette commande a déjà été décrite.

cas_suppose : donne les sous-buts que l'utilisateur a déclaré comme étant impossibles ou triviaux. Cette commande a déjà été décrite.

down <pos> : à partir du nœud courant, permet de rendre le nœud *<pos>* comme nouveau nœud courant. Cette commande a déjà été décrite.

goto <pos> : permet de rendre le nœud *<pos>* comme nouveau nœud courant. Cette commande a déjà été décrite.

up : rend le nœud père du nœud courant comme nouveau nœud courant. Cette commande a déjà été décrite.

visu_arbre <pos> : visualisation partielle ou totale de l'état actuel de la démonstration. Cette commande a déjà été décrite.

visu_feuille : visualisation des feuilles de l'arborescence de démonstration. Cette commande a déjà été décrite.

visu_nœud <pos> : visualisation du sous-but présent au nœud *<pos>*. Cette commande a déjà été décrite.

6.6.7.4- Gestion de l'environnement de démonstration

Les commandes suivantes sont relatives à l'environnement de démonstration.

ajout_equation : ajout interactif d'une nouvelle équation dans l'environnement. Si cette commande permet de débloquer une situation, il faudra rajouter cette équation dans la spécification initiale et faire à nouveau la preuve complète.

ajout_schema_induc : déclaration interactive d'un schéma d'induction. Actuellement, aucun contrôle n'est réalisé sur la cohérence des schémas d'induction, qui doivent être fournis par l'utilisateur.

ajout_var_eqs : ajout interactif de variables nécessaires aux équations introduites par la commande *ajout_equation.*

ajout_var_theo : ajout interactif de variables utilisées par les schéma d'induction déclarés par la commande *ajout_schema_induc.*

visu_base : visualisation de l'environnement de démonstration. On obtient ainsi une liste des théorèmes à démontrer et des schémas d'induction disponibles, une liste des opérateurs utilisables avec leur profils et une liste des règles de réécriture utilisables, orientées ou non.

6.6.7.5- Aides en ligne

Voici maintenant les différentes commandes permettant d'avoir une aide en cours d'utilisation du logiciel.

help_commande_preuve : résume les différentes possibilités de démonstration offertes par OASIS. Cette commande a déjà été décrite.

help_induction : documentation relative au principe d'induction utilisé dans le système OASIS. Cette commande a déjà été décrite.

help_preuve : description des fonctions de preuve. Cette commande a déjà été décrite.

help_réécriture : description de la technique de réécriture. Cette commande a déjà été décrite.

6.6.7.6- Divers

Voici enfin quelques commandes offertes par OASIS et autorisées par l'interface.

banaliser <pos> : supprime l'étiquette précédemment attribuée au nœud *<pos>* (voir la commande *nommer*). Cette commande a déjà été décrite.

calcul : visualisation des différentes étapes de réécriture réalisées lors de la dernière commande. Cette commande a déjà été décrite.

details_off : demande à ce que les réécritures soient réalisées sans visualisation des étapes intermédiaires (voir la commande *details_on*). Cette commande a déjà été décrite.

details_on : demande de visualiser tous les pas de réécriture (voir la commande *details_off*). Cette commande a déjà été décrite.

etat_permut : indique la stratégie courante quant aux règles de réécriture permutatives. Cette commande a déjà été décrite.

impossible <pos> : décrète que le sous-but présent au nœud *<pos>* est impossible. Cette commande a déjà été décrite.

liste_commande : donne la liste des commandes offertes par l'interface.

modif_permut : permet de modifier la stratégie adoptée quant aux règles de réécriture permutatives. Cette commande a déjà été décrite.

nommer <pos> : affecte une étiquette symbolique au nœud *<pos>* (voir la commande *banaliser*). Cette commande a déjà été décrite.

permut_off : supprime la possibilité d'utiliser des règles de réécriture permutatives (voir la commande *permut_on*). Cette commande a déjà été décrite.

permut_on : active la possibilité d'utiliser des règles de réécriture permutatives (voir la commande *permut_off*). Cette commande a déjà été décrite.

PROLOG : donne le contrôle à l'interpréteur PROLOG. Il est nécessaire d'évaluer la clause *-stop* pour retourner sous le contrôle de l'interface. Cette commande a déjà été décrite.

trivial <pos> : décrète que le sous-but présent au nœud *<pos>* est trivial (voir la commande *impossible*). Cette commande a déjà été décrite.

6.7- Exemple d'utilisation du logiciel

Nous allons présenter, dans cette partie, quelques exemples d'utilisation de l'interface lors d'une session de travail sous le système LPG. On peut trouver dans [Dra 88] deux exemples complets d'utilisation des fonctionnalités (fonction de preuve et de validation) de l'interface.

6.7.1- Exemple de preuve

L'exemple de preuve présenté porte sur le type abstrait des valeurs booléennes tel qu'il est défini dans la bibliothèque prédéfinie de LPG. Voici sa spécification :

```
type Bool sortes bool
constructeurs
   vrai : -> bool
   faux : -> bool
operateurs
   non : bool -> bool
   et : (bool,bool) -> bool
   ou : (bool,bool) -> bool
variables
   b,b1,b2 : bool
axiomes
   1 : non(vrai) = faux
   2 : non(faux) = vrai
   3 : vrai et b = b
   4 : faux et b = faux
   5 : b1 ou b2 = non(non(b1) et non(b2))
modeles
   - : MC [bool operateurs vrai,et]
   - : MC [bool operateurs faux,ou]
theoremes
   1 : non(non(b)) = b
   2 : b et non(b) = faux
   3 : b ou non(b) = vrai
fin Bool
```

Ce type est défini par ses deux constructeurs. On spécifie les opérations *non*, *et* et *ou* à l'aide de cinq équations. On déclare deux modèles de la propriété *MC* qui reflète l'associativité, la réflexivité et la caractère monoïde d'opérateurs. Ces différentes propriétés sont spécifiées dans d'autres modules de la bibliothèque prédéfinie de LPG. On peut voir l'arbre de dépendance de la propriété MC dans la partie décrivant les exemples de validation. On déclare enfin trois théorèmes qui doivent être démontrés pour pouvoir être utilisés par le système.

Etant sous le contrôle de LPG, on initialise la session de démonstration par la commande offerte par l'interface *prouver*.

```
Lpg : prouver
```

On donne ensuite le nom de l'unité LPG avec laquelle on veut travailler.

```
Nom de l'unite LPG a traiter ou '-' pour stopper :
?Bool
```

Le théorème à prouver est **non(non(x)) = x**. L'opérateur *non* a été suffixé par le numéro 12 (en fait, le numéro d'ordre de cet opérateur parmi les définitions connues de LPG), la variable a été renommée en A1.

```
theoreme a prouver
     1 : non12(non12(A1)) = A1.
pour poursuivre: REDUCTION., INDUCTION., CAS_AUTOMATIQUE., CAS_MANUEL.
pour avoir des explications: HELP_PREUVE.
```

On peut demander à visualiser l'environnement de démonstration.

?visu_base

On visualise les théorèmes à démontrer sous forme de clauses Prolog, manipulées par OASIS. L'entête de clause est le nom de l'unité à partir de laquelle on retrouve les théorèmes, converti en majuscule (*BOOL*). Le corps de la clause est constitué de l'identificateur *THEO.<num>* pour indiquer que c'est un théorème dont le numéro est *<num>* et du théorème lui-même, sous forme préfixée. Les opérateurs infixés sont dénotés de la manière suivante : *('<nom>'.'$')*, les autres opérateurs sont notés plus simplement : *'<nom>'*.

La présence de *BOOL(TYPE)* est nécessaire pour des raisons de fonctionnement de OASIS.

On peut remarquer que les noms des opérateurs sont suffixés par un numéro. Cela permet pour le système OASIS de différencier les opérateurs surchargés.

On peut voir des déclarations de variables. Le corps d'une telle clause comporte trois éléments. Le premier signale la déclaration elle-même, le deuxième est l'identificateur de la variable et le troisième est la sorte de la variable.

```
Unite de travail :
BOOL(TYPE).
BOOL(VARTHEO,A1,BOOL).
BOOL(THEO.1,('=','$')('non12'('non12'(A1)),A1)).
BOOL(VARTHEO,B1,BOOL).
BOOL(THEO.2,('=','$')(('et15','$')(B1,'non12'(B1)),FAUX)).
BOOL(VARTHEO,C1,BOOL).
BOOL(THEO.3,('=','$')(('ou17','$')(C1,'non12'(C1)),VRAI)).
```

C'est à ce niveau que l'on peut trouver les définitions de schémas d'induction.

On a la liste des profiles des opérateurs utilisés, toujours sous forme de clauses Prolog. La tête de clause est le symbole *S* (symbole imposé par OASIS). Le corps de la clause est formé du nom de l'opérateur avec son domaine suivi de son co-domaine.

```
Liste des operateurs avec leur profil :
S('non12'(BOOL),BOOL).
S(('et15','$')(BOOL,BOOL),BOOL).
S(('ou17','$')(BOOL,BOOL),BOOL).
S(VRAI,BOOL).
S(FAUX,BOOL).
```

On trouve, enfin, l'ensemble des règles de réécriture que pourra utiliser le démonstrateur. Cet ensemble est constitué de paquets de clauses.

Il existe un paquet par définition d'opérateur. Dans un paquet, il y a autant de clauses qu'il y a d'équations définissant l'opérateur. La tête d'une clause est le nom de l'opérateur, infixé ou non, entre apostrophes : *'<nom_op>'*. Le corps de la clause est constitué de l'équation. Il a la même forme que celle des théorèmes vus précédemment.

Les variables des équations sont traduites sous la forme d'une étoile (*) suivie d'un numéro.

```
Definition des operateurs :
'non12'('non12'(VRAI),FAUX).
'non12'('non12'(FAUX),VRAI).
'et15'(('et15'.'$')(VRAI,*1),*1).
'et15'(('et15'.'$')(FAUX,*1),FAUX).
'et15'(('et15'.'$')(('et15'.'$')(*1,*2),*3),('et15'.'$')(*1,('et15'.'$')(*2,*3))).
'et15'(('et15'.'$')(*1,VRAI),*1).
'ou17'(('ou17'.'$')(*1,*2),'non12'(('et15'.'$')('non12'(*1),'non12'(*2)))).
'ou17'(('ou17'.'$')(('ou17'.'$')(*1,*2),*3),('ou17'.'$')(*1,('ou17'.'$')(*2,*3))).
'ou17'(('ou17'.'$')(FAUX,*1),*1).
'ou17'(('ou17'.'$')(*1,FAUX),*1).
Regles permutatives :
regles_sym(('et15'.'$')(*1,*2),('et15'.'$')(*2,*1),Bool2001).
regles_sym(('ou17'.'$')(*1,*2),('ou17'.'$')(*2,*1),Bool2002).
```

On peut retrouver les équations définissants les opérateurs et mentionnées dans la spécification (lignes 1, 2, 3, 4 et 7). On remarque également la présence d'autres équations qui proviennent de la déclaration des modèles. On peut voir ainsi deux équations définissant l'associativité des opérateurs *et* et *ou* (lignes 5 et 8) et des équations définissant le caractère monoïde des deux opérateurs par rapport aux deux constantes (lignes 9, 10 pour le *ou* par rapport à *FAUX* et 6 et 3 pour le *et* par rapport à *VRAI*; dans ce dernier cas, l'équation 3 existait déjà et n'a donc pas été générée à nouveau).

Ces équations sont orientables et peuvent être utilisées sans problème comme règles de réécriture. Deux autres équations décrivant la commutativité des opérateurs *et* et *ou* ont été introduites. Ce sont les équations permutatives provenant également de la déclaration des modèles. En OASIS, les équations permutatives sont enregistrées dans un paquet à part appelé **regles_sym**. Le corps d'une telle clause est constitué de trois parties :

- le membre gauche de l'équation,
- le membre droit de l'équation,
- l'étiquette alphanumérique de l'équation permutative qui permet de la différencier des équations traditionnelles.

Nous allons inclure dans l'environnement de démonstration un schéma d'induction sur les booléens. Ce schéma trivial traduit la règle d'inférence :

$$\forall x \in \text{Bool}, \frac{P(\text{FAUX}), P(\text{VRAI})}{P(x)}$$

Dans un premier temps, il faut ajouter les variables qui vont figurer dans le schéma.

```
?ajout_var_theo
var_theo?={<nom>{,<nom>}:<type>.}.
<b : BOOL.
```

Maintenant, on ajoute effectivement le schéma d'induction. On lui attribue arbitrairement le numéro 1.

```
?ajout_schema_induc
schema_induc?={<numero>:<expression>.}.
<1 : P(b) <= (P(FAUX) et P(VRAI)).
```

Le résultat de ces deux déclarations se traduit par l'ajout des deux clauses :

```
BOOL(VARTHEO,'b',BOOL).
BOOL(SCHEMA.1,('<=','$')(P('b'),('et','$')(P(FAUX),P(VRAI)))).
```

Le schéma d'induction étant pris en compte, on va commencer la démonstration en utilisant l'induction structurelle. Il faut donner au système le numéro du schéma que l'on désire utiliser (dans notre cas, *1*), puis le nom de la (des) variable (s) sur lesquelles portera l'induction (dans notre cas, *A1*).

```
?induction
theoreme a prouver par induction
non12(non12(A1)) = A1
quel schema d'induction ([type ou rep] numero)?<1.
     1 : P(b) <= (
                    P(FAUX)
                  et
                    P(VRAI)).
nom [s] de [s] variable [s] d'induction?
 (variables du theoreme separees par des blancs)
<A1.
```

Le système nous indique alors quels sont les sous-buts, générés par le schéma d'induction, qu'il faut démontrer. Le théorème initial, dans l'exemple, a été décomposé en deux sous-buts, du fait de la présence de la conjonction dans le schéma.

```
resultat de l'expansion du schema d'induction :
noeud numero 1 1
theoreme:
non12(non12(FAUX)) = FAUX
assertions:
pas d'assertions
noeud numero 1 2
theoreme:
non12(non12(VRAI)) = VRAI
assertions:
pas d'assertions
```

Pour chacun de ces cas, nous allons appliquer la réécriture. Le nœud courant est alors le théorème initial, la commande *reducas* va s'appliquer à ses deux fils.

```
?reducas
noeud numero 1 1
theoreme:
non12(non12(FAUX)) = FAUX
assertions:
pas d'assertions
```

```
Reduction par reecriture
temps uc = 0  secondes  78  millisecondes
resultat final:
 VRAI
le theoreme est PROUVE (dans ce cas)
nouveau noeud cree: 1 1 R
```

Le système a réussi à réécrire le premier sous-but en *VRAI*. Il considère ce cas comme démontré. On passe automatiquement au deuxième cas généré.

```
noeud numero 1 2
theoreme:
non12(non12(VRAI)) = VRAI
assertions:
pas d'assertions
Reduction par reecriture
temps uc = 0  secondes  74  millisecondes
resultat final:
 VRAI
le theoreme est PROUVE (dans ce cas)
nouveau noeud cree: 1 2 R
```

La commande **cas_restant** peut nous indiquer quelles sont les feuilles de l'arborescence non encore réduites à VRAI ou non déclarées TRIVIAL/IMPOSSIBLE. Quand tout est démontré, OASIS le signale à l'issue de cette commande. Le système nous rappelle qu'il est possible de lister les différents sous-buts qui ont été éventuellement déclarés TRIVIAL ou IMPOSSIBLE.

```
?cas_restant
cas restant a prouver:
 Bravo! le theoreme est entierement prouve
 vous pouvez lister les cas supposes impossibles ou triviaux
 par CAS_SUPPOSE
```

La démonstration du premier théorème est maintenant terminée. On nous propose de démontrer les autres théorèmes, s'il en reste.

```
Voulez-vous continuer avec les autres theoremes (o/n) ?
```

La démonstration des deux autres théorèmes est tout aussi facile que celle du premier. Quand tout les théorèmes ont été traités (démontrés ou non, dans le cas où on abandonne la démonstration), on peut, si on le désire, entamer une nouvelle session de démonstration.

```
Voulez-vous realiser une autre session de preuve (o/n) ?
```

En fonction de la réponse, soit on reste sous le contrôle de l'interface, soit on retourne sous le contrôle de LPG.

6.7.2- Exemple de validation

Une session de validation fait intervenir en dernier lieu diverses sessions de démonstration analogues à celle vue précédemment, mais généralement plus compliquées. Avant de commencer ces différentes démonstrations, l'interface analyse l'unité LPG que l'on désire valider afin de détecter toutes les autres entités (morphismes, modèles et unités) à valider.

L'exemple présenté consiste à valider l'enrichissement *Op_sur_nat* à partir d'une bibliothèque contenant le type *Bool* ainsi que le type *Naturel* qui ont été déjà validés. Les théorèmes de *Naturel* sont démontrés ainsi que ceux de *Bool*.

On débute la session en actionnant la commande de validation.

```
Lpg :valider
Nom de l'unite LPG a traiter ou '-' pour stopper :
?Op_sur_nat
```

Le système nous propose les différentes possibilités. On peut visualiser la liste des différentes entités à valider pour que l'unité *Op_sur_nat* soit valide. Le système peut essayer de valider automatiquement tout ce qu'il peut. En effet, une unité LPG qui n'importe aucune définition et qui ne déclare ni morphisme ni modèle est valide par définition, comme c'est le cas par exemple de la propriété *Tyfo*. D'autres cas semblables sont pris en compte par l'interface. On peut également débuter les différentes démonstrations que l'utilisateur aura généralement à réaliser. Enfin, il est possible d'arrêter le traitement.

```
Voulez-vous :
    - obtenir la liste de ce qu'il faut valider ---> (l) ?
    - lancer une validation automatique ----------> (a) ?
    - lancer une validation non automatique -------> (v) ?
    - stopper ------------------------------------> (-) ?
```

On va demander à visualiser tout ce qu'il nous faut valider. Cette information peut être présentée sous la forme d'une liste d'entités à valider ou d'une arborescence dont chaque nœud contient une entité éventuellement déjà validée.

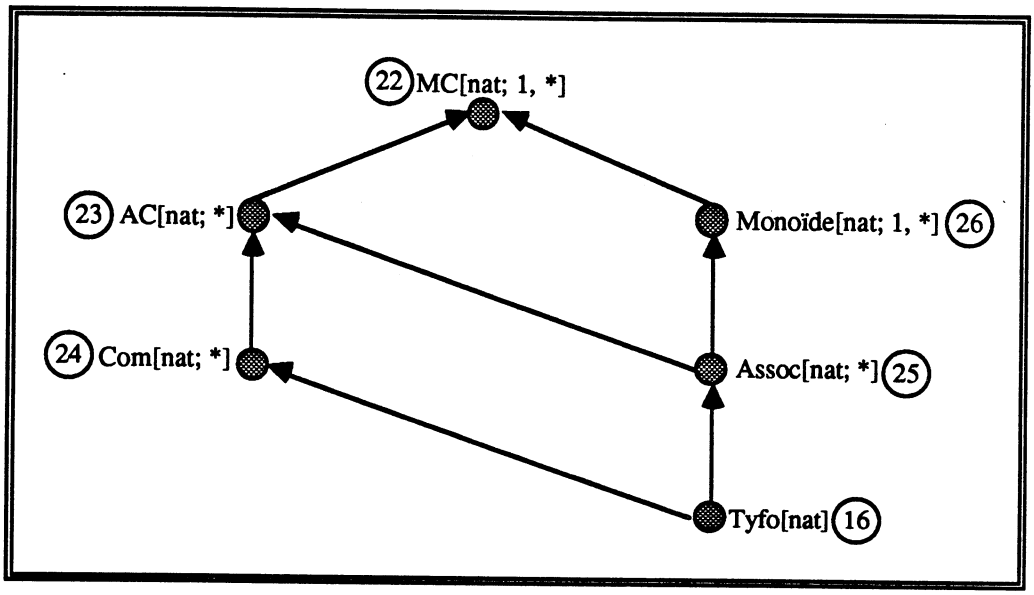
La liste présente l'avantage d'être concise car on ne montre que ce qu'il faut valider et de manière unique. L'arborescence indique plus précisément que, pour que l'entité présente à un certain nœud soit valide, il faut que les entités figurant dans tous les nœuds fils soient valides. On trouvera notamment dans cette arborescence aussi bien les entités déjà validées que celles qui restent à valider.

Cette arborescence représente en quelque sorte un graphe de dépendance. De ce fait, il est possible de trouver plusieurs branches de l'arborescence dupliquées, comme par exemple, la validation de la propriété *Equality* qui implique la validation de l'unité *Bool* et la validation de l'enrichissement *Op_sur_nat* qui implique également la validation de *Bool*.

On peut toutefois obtenir une arborescence plus ou moins détaillée en fixant, au départ la profondeur maximum d'exploration. En donnant une grande valeur (1000, par exemple), on est presque sûr d'obtenir la totalité de l'arborescence. En se limitant aux premiers niveaux, cette représentation peut fournir des informations intéressantes concernant les dépendances.

Dans un premier temps, on demande simplement le liste ordonnée des différentes entités à valider. L'ordre proposé de validation doit être respecté. Il faut d'abord valider le modèle numéro 16 de la propriété *Tyfo*, puis le modèle 25 de la propriété *Assoc*, etc.

On peut voir, par exemple sur le schéma suivant les dépendances entre les propriétés *MC*, *AC*, *Com*, *Monoïde*, etc. avec les numéros des modèles déclarés par l'unité *Op_sur_nat* ainsi que les morphismes définissant ces modèles.



Arbre de dépendance de la propriété MC

```
?l
Voulez-vous la liste, l'arborescence ou stopper (l/a/-) ?
?l
LISTE DE VALIDATION :
═══════════════════
Le modele 16, de la propriete Tyfo
Le modele 25, de la propriete Assoc
Le modele 26, de la propriete Monoide
Le modele 24, de la propriete Com
Le modele 23, de la propriete AC
Le modele 22, de la propriete MC
Le modele 20, de la propriete Assoc
Le modele 21, de la propriete Monoide
Le modele 19, de la propriete Com
Le modele 18, de la propriete AC
Le modele 17, de la propriete MC
Le morphisme 8 (Tyfo ——> Egalite)
La propriete Egalite
Le modele 15, de la propriete Egalite
Le morphisme 9 (Egalite ——> Ordre_partiel)
La propriete Ordre_partiel
Le modele 14, de la propriete Ordre_partiel
La propriete Ordre_total
Le modele 13, de la propriete Ordre_total
Le type ou enrichissement Op_sur_nat
Voulez-vous la liste, l'arborescence ou stopper (l/a/-) ?
```

Dans un deuxième temps, on va visualiser l'arborescence de validation. La taille de cette arborescence étant particulièrement grande, on ne va présenter ici que le début.

Chaque ligne représente un nœud dans l'arborescence. Le numéro indique la profondeur de ce nœud (la profondeur de la racine est 0). On trouve ensuite le type d'entité contenu dans le nœud (unité, morphisme ou modèle), éventuellement le

nom de l'entité (pour les unités). Si cette entité est déjà validée, le symbole *v* figurera entre parenthèses. Pour un morphisme, on trouvera le numéro du morphisme ainsi que sa source et sa cible. Pour un modèle, on aura le numéro du modèle et le nom de la propriété dont il est issu.

Dans l'exemple ci-dessous, on demande une profondeur maximum d'exploration de l'arborescence égale à 987. Pour l'exemple, cela représente la totalité de l'arborescence dont la profondeur maximale est en fait égale à 12, [Dra 88]. On ne présente toutefois que le début de l'arborescence du fait de sa grandeur très importante.

On remarque sur cet exemple que seule l'unité LPG *Op_sur_nat* doit être validée. Les autres entités ont été précédemment validées ([Dra 88]).

```
?a
Profondeur maximum d'exploration de l'arbre ?
?987
ARBORESCENCE DE VALIDATION (profondeur maximum = 987) :
═══════════════════════════════════════════════════════════
 0 |Le type ou enrichissement Op_sur_nat implique la validation de :
 1 |__Le type ou enrichissement Naturel (v) implique la validation de :
 1 |__Le type ou enrichissement Bool (v) implique la validation de :
 2 |____Le modele (v) 1, de la propriete MC implique la validation de :
 3 |______La propriete MC (v) implique la validation de :
 4 |________La propriete AC (v) implique la validation de :
 5 |__________La propriete Com (v) implique la validation de :
 6 |____________Le morphisme (v) 1 (Tyfo ---> Com) implique la validation de :
 7 |______________La propriete Tyfo (v) implique la validation de :
 5 |__________La propriete Assoc (v) implique la validation de :
 6 |____________Le morphisme (v) 2 (Tyfo ---> Assoc) implique la validation de :
 7 |______________La propriete Tyfo (v) implique la validation de :
 4 |________La propriete Monoide (v) implique la validation de :
 5 |__________La propriete Assoc (v) implique la validation de :
 6 |____________Le morphisme (v) 2 (Tyfo ---> Assoc) implique la validation de :
 7 |______________La propriete Tyfo (v) implique la validation de :
 3 |______Le modele (v) 2, de la propriete AC implique la validation de :
 4 |________La propriete AC (v) implique la validation de :
 5 |__________La propriete Com (v) implique la validation de :
 6 |____________Le morphisme (v) 1 (Tyfo ---> Com) implique la validation de :
 7 |______________La propriete Tyfo (v) implique la validation de :
 5 |__________La propriete Assoc (v) implique la validation de :
 6 |____________Le morphisme (v) 2 (Tyfo ---> Assoc) implique la validation de :

.
.
.
```

On demande maintenant à entamer le processus de validation non automatique. Le système dans un premier temps essaie tout de même de valider automatiquement tout ce qu'il pourra. Ensuite, il propose à l'utilisateur le reste des validations.

```
?v
Et un morphisme de valider, un !
C'est le numero : 8 entre les proprietes Tyfo et Egalite.
```

```
La proriete Egalite vient d'etre validee sous vos yeux...
Ben mince, alors !!! Sans meme m'en rendre compte,
je viens de valider le modele numero : 16 de la propriete Tyfo.
```

On peut remarquer que les propriétés *Tyfo* et *Egalite* ont été validées automatiquement.

Le système va maintenant proposer les différentes entités qu'il n'a pas pu valider. L'utilisateur peut décider d'accepter de tenter la validation, de la refuser mais continuer avec le reste (auquel cas, la validation générale échouera) ou tout simplement arrêter le processus de validation (dans ce cas, la validation générale échoue également).

Pour la validation d'un modèle, le système précise le numéro du modèle et le nom de la propriété d'origine. Pour un morphisme, on donne également le numéro ainsi que les propriétés source et cible. Quand il s'agit d'une unité, la validation est faite automatiquement. En effet, la validation d'une unité revient a démontrer tous les théorèmes générés par les différentes déclarations de modèles et morphismes. Quand toutes ces démonstrations sont faites l'unité est alors, par définition, validée.

Maintenant commence donc la partie de validation non automatique, c'est à dire que l'utilisateur doit réaliser diverses sessions de démonstrations afin de vérifier les différentes conditions imposées par la sémantique du langage. La session toute entière de validation étant beaucoup trop longue, il n'est nullement question de la faire figurer dans ce document. Le lecteur intéressé peut se rapporter à [Dra 88] pour y trouver une description plus détaillée de cette session.

On ne présente de ce fait que la session de démonstration qui concerne l'associativité de l'opérateur de multiplication sur les naturels :

$$\forall x,y,z \in nat,\ x*(y*z)=(x*y)*z$$

En effet, la première entité à valider est le modèle numéro *25* de la propriété *Assoc*. Bien que l'utilisateur n'a pas déclaré explicitement un tel modèle, ce dernier est déclaré automatiquement lors de la déclaration du modèle numéro *22* de la propriété *MC*. Comme on peut voir sur la précédente figure (*Arbre de dépendance de la propriété MC*), cette propriété dépend de la propriété *Monoïde* qui dépend elle-même de la propriété *Assoc*. Un modèle déclaré de *MC* génère donc automatiquement la déclaration d'un modèle de *Assoc*.

```
Voulez-vous valider le modele numero 25 de la propriete Assoc ou stopper totalement l
validation (o/n/-) ?
?o
theoreme a prouver
     1 : A1 *43 B2 *43 C3 = A1 *43 (B2 *43 C3).
pour poursuivre: REDUCTION., INDUCTION., CAS_AUTOMATIQUE., CAS_MANUEL.
pour avoir des explications: HELP_PREUVE.
?details_on
```

Cette commande va permettre de visualiser toutes les étapes de réécriture durant la session de preuve. A tout moment, on peut annuler son effet en utilisant la commande *details_off*.

On demande maintenant à regarder le contenu courant de l'environnement de démonstration.

```
visu_base
Unite de travail :
OP_SUR_NAT(TYPE).
OP_SUR_NAT(VARTHEO,A1,'nat').
OP_SUR_NAT(VARTHEO,B2,'nat').
OP_SUR_NAT(VARTHEO,C3,'nat').
OP_SUR_NAT(THEO.1,('=','$')(('*43','$')(('*43','$')(A1,B2),C3),('*43','$')(A1,('*43','$
')(B2,C3)))).
Liste des operateurs avec leur profil :
S(('=','$')('nat','nat'),BOOL).
S(('*43','$')('nat','nat'),'nat').
S('succ39'('nat'),'nat').
S(('+40','$')('nat','nat'),'nat').
S('z','nat').
Definition des operateurs :
'*43'(('*43','$')('z',*1),'z').
'*43'(('*43','$')('succ39'(*1),*2),('+40','$')(*2,('*43','$')(*1,*2))).
'*43'(('*43','$')(*1,'z'),'z').
'*43'(('*43','$')(*1,'succ39'(*2)),('+40','$')(('*43','$')(*1,*2),*1)).
'*43'(('*43','$')(*1,('+40','$')(*2,*3)),('+40','$')(('*43','$')(*1,*2),('*43','$')(*1,
*3))).
'*43'(('*43','$')(('+40','$')(*1,*2),*3),('+40','$')(('*43','$')(*1,*3),('*43','$')(*2,
*3))).
'+40'(('+40','$')('z',*1),*1).
'+40'(('+40','$')('succ39'(*1),*2),'succ39'(('+40','$')(*1,*2))).
'+40'(('+40','$')(*1,'z'),*1).
'+40'(('+40','$')(*1,'succ39'(*2)),'succ39'(('+40','$')(*1,*2))).
Regles permutatives :
```

On remarque la présence des équations définissant la multiplication sur les naturels (lignes numéro 1 et 2) ainsi que les équations définissant l'addition (lignes numéro 7 et 8) du fait que la multiplication est définie en utilisant l'addition.

Les autres lignes proviennent de la déclaration, et démonstration antérieure, de théorèmes déclarés dans la spécification *Naturel*. On peut trouver ainsi quatre équations concernant la multiplication (lignes 3, 4, 5 et 6) qui traduisent entre autres la distributivité de la multiplication sur l'addition. Deux équations relatives à l'addition provenant également de démonstrations antérieures ont été inclues dans cet environnement (lignes 9 et 10).

Afin de pouvoir réaliser les démonstrations, nous allons ajouter un schéma d'induction structurel sur les naturels. Il correspond à la règle d'inférence :

$$\forall x,y \in \text{nat}, \quad \frac{P(z),\ P(y) \Rightarrow P(succ(y))}{P(x)}$$

Le symbole *z* représente la constante zéro des entiers naturels[1].

[1] Dans la bibliothèque prédéfinie de LPG, On ne trouve pas le constructeur "zéro" dans la signature des naturels. Le chiffre "0" représente alors le constructeur manquant des naturels. Pour OASIS, un chiffre ne peut pas être un identificateur d'opérateur. Il a donc été décidé de prendre le symbole "z".

Il est nécessaire, pour cela, de déclarer les deux variables utilisées dans le schéma.

```
ajout_var_theo
var_theo?={<nom>{,<nom>}:<type>.}.
<x, y : nat.
<.
ajout_schema_induc
```

Grâce à la précédente visualisation de l'environnement de démonstration, on peut identifier précisément les identificateurs des constructeurs des naturels (*z* et *succ39*) qui seront utilisés dans le schéma. L'identificateur du deuxième constructeur des naturels est suffixé par *39* (en fait, c'est le trente neuvième opérateur dans les définitions de LPG).

Dans une version future de ce logiciel, on pourrait facilement prendre en compte la génération automatique de tels schémas d'induction et donc on n'aurait plus à se soucier de ce genre de subtilités.

```
schema_induc?={<numero>:<expression>.}.
<1 : P(x) <= (P(z) et (P(y) => P(succ39(y)))).
<.
```

On demande, pour commencer la démonstration, à réaliser l'induction sur la variable *A1*.

```
induction
theoreme a prouver par induction
A1 *43 B2 *43 C3 = A1 *43 (B2 *43 C3)
quel schema d'induction ([type ou rep] numero)?<1.
     1 : P(x) <= (
                    P(z)
                  et
                    (
                        P(y)
                      =>
                        P(succ39(y)))).
nom [s] de [s] variable [s] d'induction?
 (variables du theoreme separees par des blancs)
<A1.
resultat de l'expansion du schema d'induction :
```

Suivant la règle d'inférence, le théorème initial est décomposé en deux sous-buts, l'étape triviale et l'étape inductive, correspondant aux nœuds numéro *1 1* et *1 2* de l'arborescence.

```
noeud numero 1 1
theoreme:
z *43 B2 *43 C3 = z *43 (B2 *43 C3)
assertions:
pas d'assertions
noeud numero 1 2
theoreme:
(y *43 B2 *43 C3 = y *43 (B2 *43 C3)) => (succ39(y) *43 B2 *43 C3 = succ39(y) *43 (B2
*43 C3))
assertions:
pas d'assertions
```

Le cas trivial se démontre généralement facilement par simple réécriture (utilisation de la commande *reduction*).

```
?reduction 1 1
noeud numero 1 1
theoreme:
z *43 B2 *43 C3 = z *43 (B2 *43 C3)
assertions:
pas d'assertions
Reduction par reecriture
z *43 B2 ---> z
z *43 C3 ---> z
z *43 (B2 *43 C3) ---> z
z = z -------> VRAI
temps uc = 0 secondes 403 millisecondes
resultat final:
 VRAI
le theoreme est PROUVE (dans ce cas)
```

Les étapes de réécriture sont représentées par *<membre gauche> <flèche> <membre droit>*. La flèche peut être de trois types :

- m_1---> m_2 : réécriture simple, m_1 se réécrit en m_2 qui pourra se réécrire à nouveau,
- m_1 ------> m_2 : réécriture globale de m_1 en m_2 qui, lui, est irréductible,
- m_1 ----- ruse -----> m_2 : réécriture directe par mémorisation d'un calcul antérieur.

```
nouveau noeud cree: 1 1 R
```

Pour l'étape inductive, on va tout d'abord prendre en compte l'hypothèse d'induction comme une assertion en utilisant la commande *cas_automatique*. Un nouveau nœud est généré (*1 2 1*).

```
?cas_automatique 1 2
noeud numero 1 2
theoreme:
(y *43 B2 *43 C3 = y *43 (B2 *43 C3)) => (succ39(y) *43 B2 *43 C3 = succ39(y) *43 (B2
*43 C3))
assertions:
pas d'assertions
generation automatique des cas suivants:
noeud numero 1 2 1
theoreme:
succ39(y) *43 B2 *43 C3 = succ39(y) *43 (B2 *43 C3)
assertions:
y *43 B2 *43 C3 == y *43 (B2 *43 C3)
```

Et on va appliquer la réécriture sur le résultat. L'assertion est considérée comme une règle de réécriture ayant une priorité supérieure à celle de toutes les autres règles.

```
reducas
noeud numero 1 2 1
theoreme:
succ39(y) *43 B2 *43 C3 = succ39(y) *43 (B2 *43 C3)
assertions:
y *43 B2 *43 C3 == y *43 (B2 *43 C3)
```

```
Reduction par reecriture
succ39(y) *43 B2 ---> B2 +40 (y *43 B2)
B2 +40 (y *43 B2) *43 C3 ---> B2 *43 C3 +40 (y *43 B2 *43 C3)
y *43 B2 *43 C3 ---> y *43 (B2 *43 C3)
B2 +40 (y *43 B2) *43 C3 -------> B2 *43 C3 +40 (y *43 (B2 *43 C3))
succ39(y) *43 (B2 *43 C3) ---> B2 *43 C3 +40 (y *43 (B2 *43 C3))
B2 *43 C3 +40 (y *43 (B2 *43 C3)) = B2 *43 C3 +40 (y *43 (B2 *43 C3)) -------> VRAI
temps uc = 1 secondes 60 millisecondes
resultat final:
 VRAI
le theoreme est PROUVE (dans ce cas)
nouveau noeud cree: 1 2 1 R
```

Le deuxième sous-but vient d'être démontré. Le lecteur curieux peut suivre les étapes de réécriture en s'aidant des règles contenues dans l'environnement de démonstration.

Nous allons maintenant vérifier qu'il n'y a plus rien à prouver (tous les sous-buts ont été démontrés), ce qui s'effectue par la commande *cas_restant.*

```
cas_restant
cas restant a prouver:
 Bravo! le theoreme est entierement prouve
 vous pouvez lister les cas supposes impossibles ou triviaux
 par CAS_SUPPOSE
```

Vérifions, maintenant, qu'il n'y a pas eu de cas supposés trivial ou impossible.

```
?cas_suppose
cas supposes impossibles:
 pas de cas supposes impossibles
cas supposes triviaux:
 pas de cas supposes triviaux
```

On peut enfin visualiser l'arborescence de démonstration en utilisant la commande *visu_arbre*. L'argument *1* de la commande indique que l'on veut voir l'arborescence à partir de la racine. Cette arborescence est présentée par niveau. Pour chaque nœud, on retrouve son numéro, le but qui lui est associé et les assertions correspondantes quand il y en a.

```
?visu_arbre 1
noeud numero 1  THEOREME_INITIAL
theoreme:
A1 *43 B2 *43 C3 = A1 *43 (B2 *43 C3)
assertions:
pas d'assertions
```

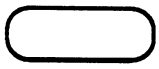
```
noeud numero 1 1
theoreme:
z *43 B2 *43 C3 = z *43 (B2 *43 C3)
assertions:
pas d'assertions
noeud numero 1 2
theoreme:
(y *43 B2 *43 C3 = y *43 (B2 *43 C3)) => (succ39(y) *43 B2 *43 C3 = succ39(y) *43 (B2
*43 C3))
assertions:
```

```
pas d'assertions
```

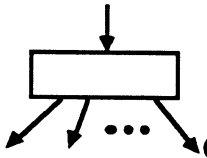
```
noeud numero 1 1 R
theoreme:
VRAI
assertions:
pas d'assertions
noeud numero 1 2 1
theoreme:
succ39(y) *43 B2 *43 C3 = succ39(y) *43 (B2 *43 C3)
assertions:
y *43 B2 *43 C3 == y *43 (B2 *43 C3)
```

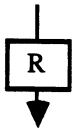
```
noeud numero 1 2 1 R
theoreme:
VRAI
assertions:
y *43 B2 *43 C3 == y *43 (B2 *43 C3)
```

On peut également représenter cette démonstration par un graphique plus parlant. Les symboles utilisés ont la signification suivante.

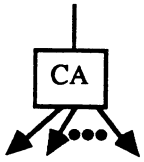

représente un nœud de l'arborescence de preuve. A l'intérieur figure le but ou sous-but qu'il faut démontrer, on peut y voir également les assertions. Ce sont des lignes commençant par *H* :.

Les arcs de l'arborescence sont de plusieurs types selon qu'ils produisent des nœuds par induction, réduction ou bien par raisonnement par cas.

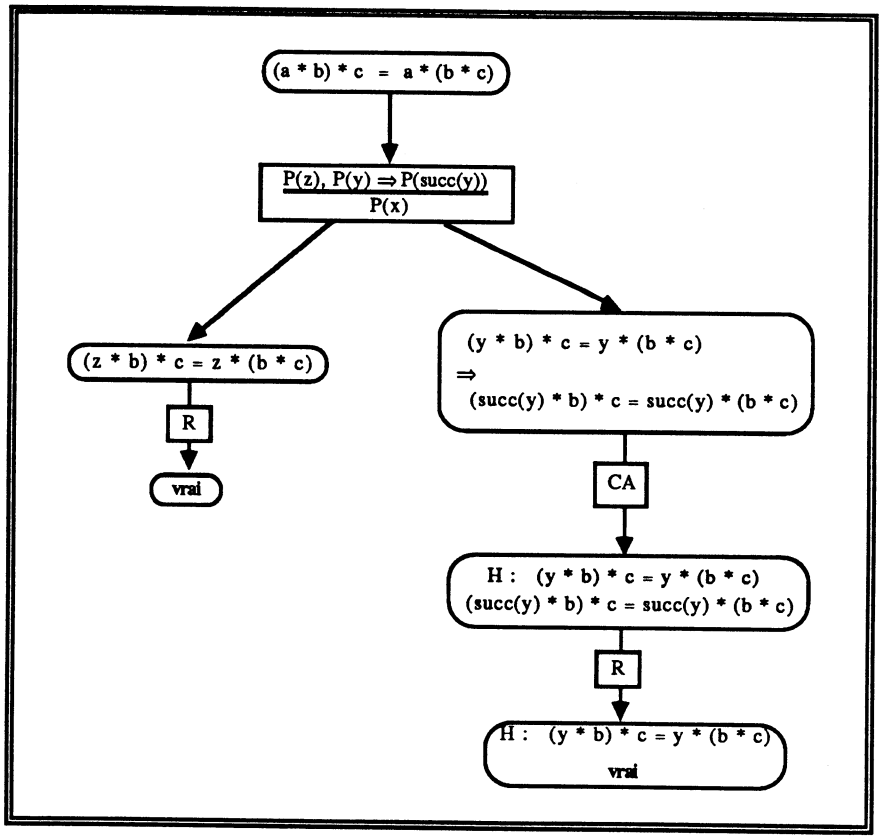

Ce symbole indique une étape d'induction. Pour démontrer le nœud père, il faut démontrer tous les nœuds fils. Le rectangle contient la règle d'inférence utilisée.



On passe du père au fils par simple réécriture.



Ce type d'arc indique une décomposition par cas automatique.



Arbre de preuve de ((a*b)*c = a*(b*c))

```
?-
```

On décide maintenant que cette session de démonstration est terminée. Si il reste d'autres validations à réaliser, le système propose la suivante.

```
Voulez-vous valider le modele numero 26 de la propriete Monoide ou stopper totalement
la validation (o/n/-) ?
```

Pour valider réellement l'unité *Op_sur_nat*, il faut répondre *oui* (*o*) à cette question et ainsi de suite pour chacune des validations proposées. L'ensemble de ces sessions de démonstrations se trouve dans [Dra 88].

7- Vers une algèbre de preuves

7.1- Introduction

Après avoir utilisé plusieurs fois l'implémentation qui a fait suite à l'étude de la notion de validation sémantique, nous avons été amené à considérer diverses fonctionnalités que l'on pourrait souhaiter trouver dans un atelier de démonstration. Certaines de ces fonctionnalités, comme par exemple la réécriture sont relativement courantes. D'autres, par exemple la réutilisation de schéma de preuves qui ont déjà été réalisées, le sont moins.

Ce concept de réutilisation de schéma de preuves conduit directement à la notion d'algèbre de preuves dont l'étude est en fait suggérée dans cette partie.

Le but de ce chapitre est donc de décrire les diverses fonctionnalités, tout en tenant compte de cette notion d'algèbre de preuves, que l'on peut souhaiter avoir dans un atelier de démonstration en utilisant le formalisme de LPG. Nous allons voir ainsi les motivations qui ont conduit à ce travail. Nous verrons alors très rapidement ce qui se passe en LCF. Nous donnerons enfin différentes définitions permettant de spécifier les fonctionnalités de notre atelier de démonstration.

7.2- Motivations

La session de validation qui est à l'origine de l'exemple du précédent chapitre a donné lieu à plusieurs démonstrations :

- $\forall a, b, c \in Nat,\ (a*b)*c = a*(b*c)$ (1)

associativité de la multiplication entre les naturels,

- $\forall a \in Nat,\ 1*a = a$ (2)

1 est élément neutre à gauche pour la multiplication entre les naturels,

- $\forall b \in Nat,\ b*1 = b$ (3)

1 est élément neutre à droite pour la multiplication entre les naturels,

- $\forall a, b, c \in Nat,\ (a+b)+c = a+(b+c)$ (4)

associativité de l'addition entre les naturels,

- $\forall a \in Nat,\ 0+a = a$ (5)

0 est élément neutre à gauche pour l'addition entre les naturels,

- $\forall b \in Nat,\ b+0 = b$ (6)

0 est élément neutre à droite pour l'addition entre les naturels,

- $\forall a, b \in Nat,\ a+b = b+a$ (7)

l'addition entre les naturels est commutative,

- $\forall a, b \in Nat,\ a*b = b*a$ (8)

la multiplication entre les naturels est commutative,

- $\forall a \in Nat,\ a=a$ (9)

l'égalité (l'opérateur =) entre les naturels est réflexive,

• $\forall b, c \in Nat, (b=c) \Rightarrow (c=b)$ (10)

l'égalité (l'opérateur =) entre les naturels est symétrique,

• $\forall c, d, f \in Nat, (d=c \text{ et } c=f) \Rightarrow (d=f)$ (11)

l'égalité (l'opérateur =) entre les naturels est transitive,

• $\forall a \in t, a=a$ (12)

opérateur = réflexif dans une théorie équationnelle (on ne connait pas à priori la sorte *t*),

• $\forall b, c \in t, (b=c) \Rightarrow (c=b)$ (13)

opérateur = symétrique dans une théorie équationnelle,

• $\forall c, d, f \in t, (d=c \text{ et } c=f) \Rightarrow (d=f)$ (14)

opérateur = symétrique dans une théorie équationnelle. Cette démonstration n'a pas été réussie en utilisant l'outil de démonstration automatique. Il a fallu la réaliser manuellement du fait de l'utilisation de règles de réécritures non orientables,

• $\forall a \in Nat, a<=a$ (15)

cet opérateur de comparaison (<=) sur les naturels est réflexif,

• $\forall b, c \in Nat, (b<=c \text{ et } c<=b) \Rightarrow b=c$ (16)

cet opérateur de comparaison (<=) sur les naturels est anti-symétrique,

• $\forall d, e, f \in Nat, (d<=e \text{ et } e<=f) \Rightarrow (d<=f)$ (17)

cet opérateur de comparaison (<=) sur les naturels est transitif et

• $\forall a, b \in Nat, (a<=b) \text{ ou } (b<=a)$ (18)

tout naturel peut être comparé à tout autre naturel (<= est une relation d'ordre totale).

Les détails de toutes ces démonstrations sont visibles dans [Dra 88].

Au fur et à mesure que l'on réalise ces démonstrations, on peut s'apercevoir que la démarche générale est relativement semblable pour toutes les preuves. On peut mettre ce fait en évidence en utilisant la représentation schématique, introduite dans le chapitre précédent, de l'arborescence de chacune des preuves.

On peut s'apercevoir facilement sur les trois représentations graphiques suivantes que les démonstrations des théorèmes numéro :

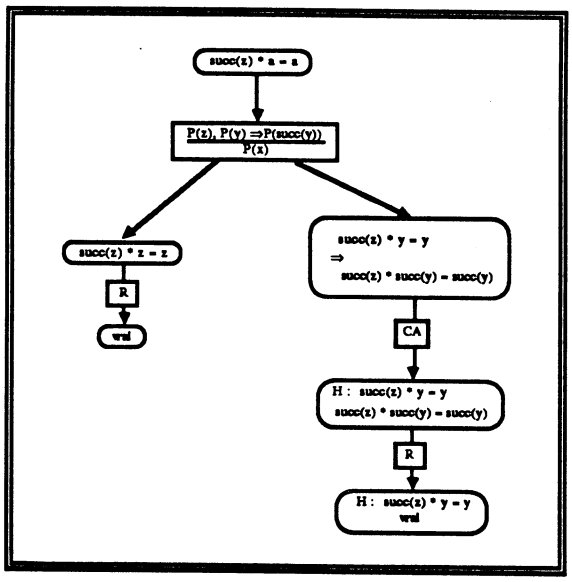
• 2, élément neutre,
• 4, associativité et
• 7 commutativité

sont tout à fait semblables.

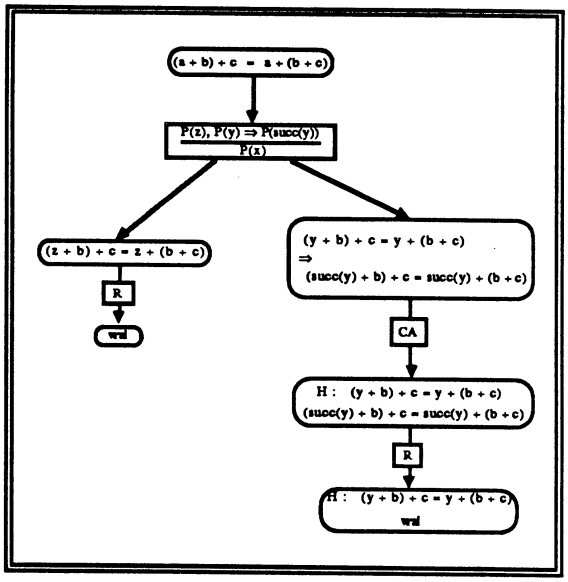
Cette même démarche de démonstration est utilisée pour les théorèmes numéro :

• 1 associativité, comme on l'a vu au chapitre précédent,
• 3 élément neutre,
• 9 et 15, réflexivité.

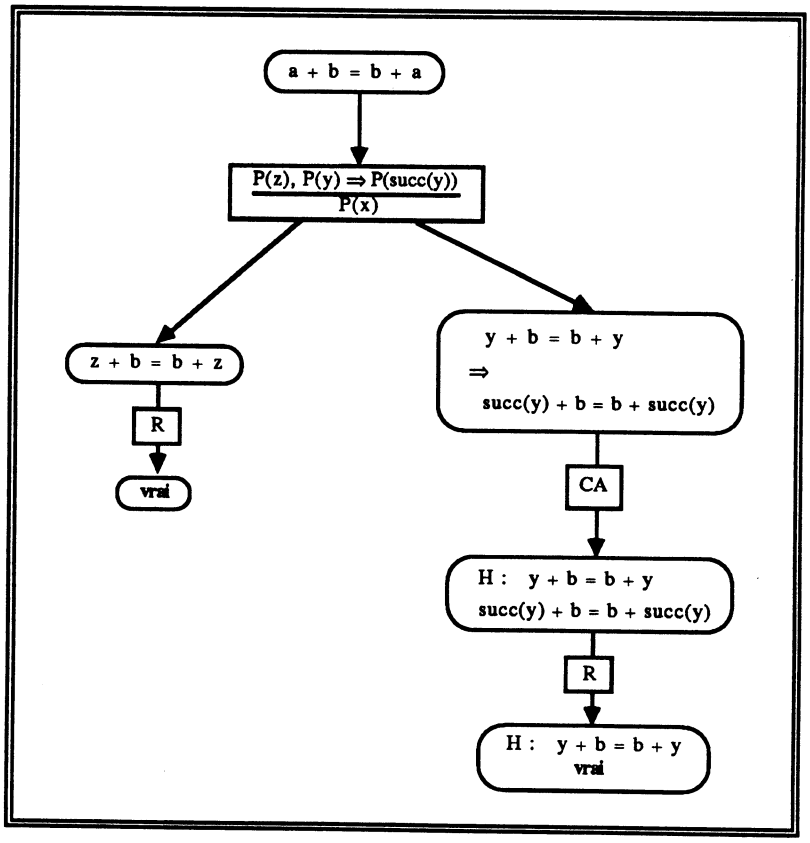
Nous ne donnons pas ici les graphiques représentant les démonstrations de ces théorèmes car ils n'apporteraient rien pour la compréhension générale.



Arbre de preuve de (1*a = a)



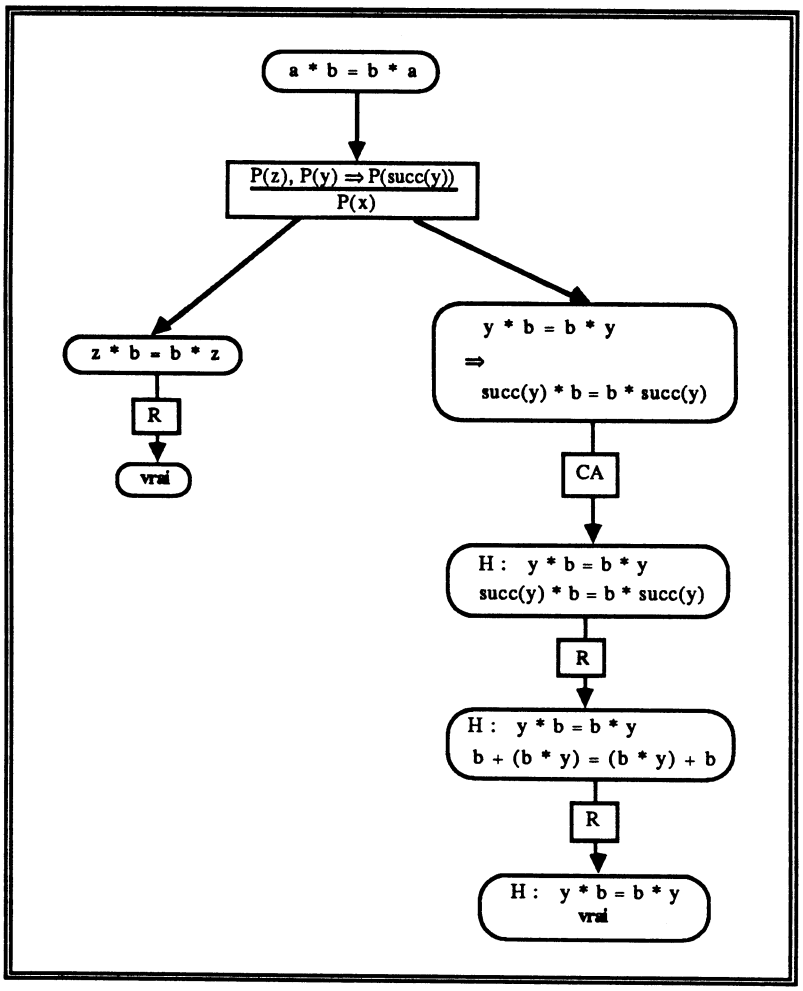
Arbre de preuve de ((a+b)+c = a+(b+c))



Arbre de preuve de (a+b = b+a)

En ce qui concerne le théorème numéro 8, commutativité de la multiplication, l'application d'un tel schéma n'a pas été suffisant. Il a été nécessaire d'utiliser les règles non orientables pour sa démonstration.

On obtient ainsi le schéma suivant dans lequel on peut remarquer deux étapes de réécriture successives.

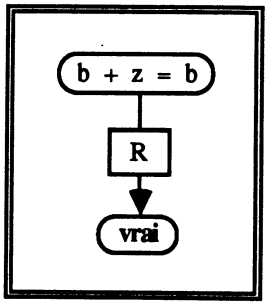


Arbre de preuve de (a*b = b*a)

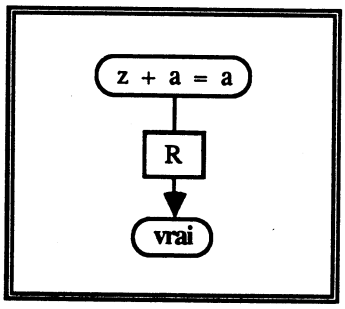
Cette analogie entre les squelettes des démonstrations se retrouve pour les théorèmes numéro :

- 5, élément neutre à gauche,
- 6, élément neutre à droite, et
- 12, réflexivité.

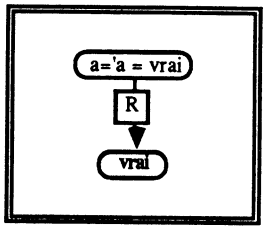
On obtient, cette fois-ci, des schémas très simples du genre :



Arbre de preuve de (b+0 = b)



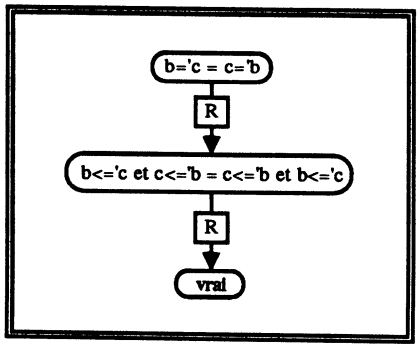
Arbre de preuve de (0+a = a)



Arbre de preuve de (a='a = vrai)

Pour le théorème équationel numéro 13, symétrie, il a fallu prendre en compte des règles non orientables.

D'où le squelette suivant dans lequel on remarque également deux étapes de réécritures successives.



Arbre de preuve de (b='c = c='b) dans une théorie équationnelle

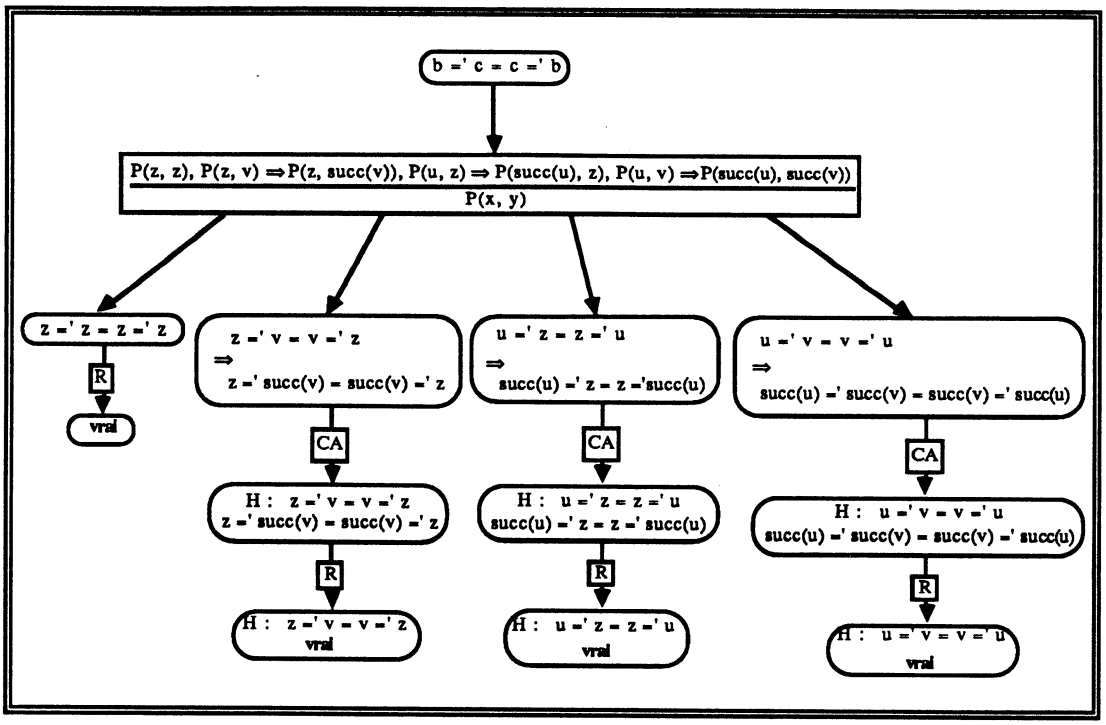
Les squelettes de démonstrations peuvent aussi être plus compliqués. Le degré de complexité peut se mesurer en fonction du nombre de sous-buts qu'il faut résoudre.

Dans cette session de validation, ce degré est fonction de la complexité du schéma d'induction utilisé (en fait, le nombre de variables sur lesquelles on fait porter l'induction). On peut ainsi obtenir des squelettes avec quatre sous-buts à résoudre.

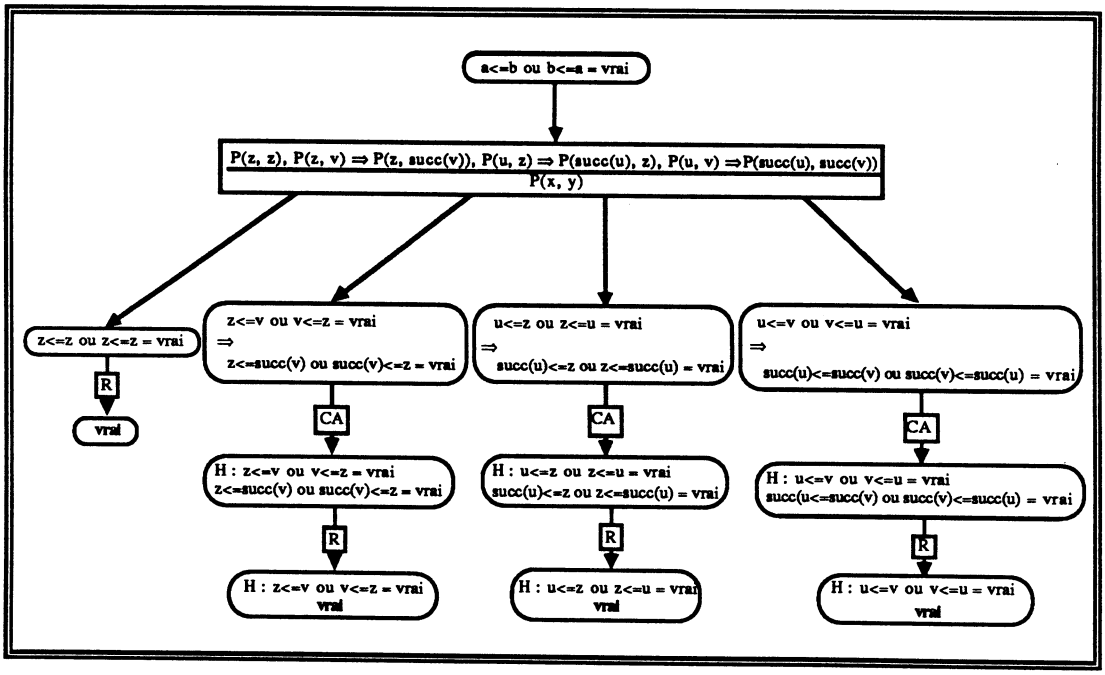
C'est le cas rencontré pour les théorèmes numéro :

- 10, symétrie de l'égalité,
- 16, anti-symétrie de la relation d'ordre, dont on ne donnera pas dans un soucis de clarté le schéma ici, et
- 18, relation d'ordre total.

On peut également remarquer l'analogie quant aux squelettes de ces démonstrations.



Arbre de preuve de (b='c = c='b)
dans une théorie inductive

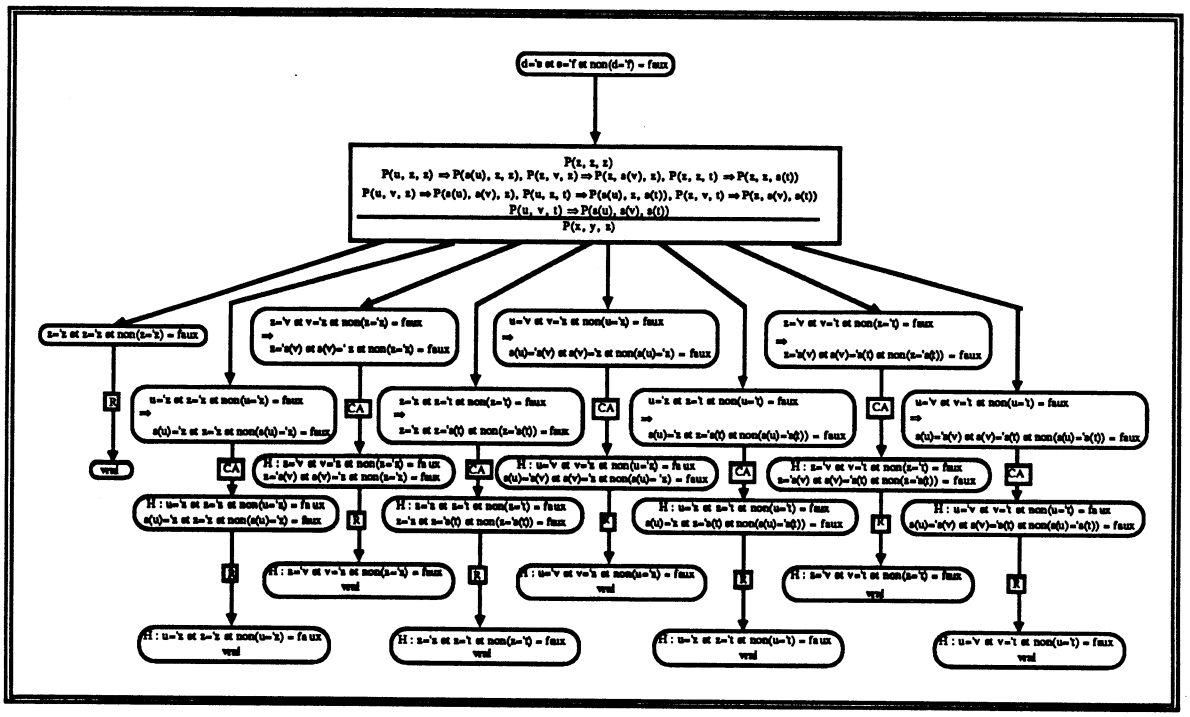


Arbre de preuve de (a<=b ou b<=a = vrai)

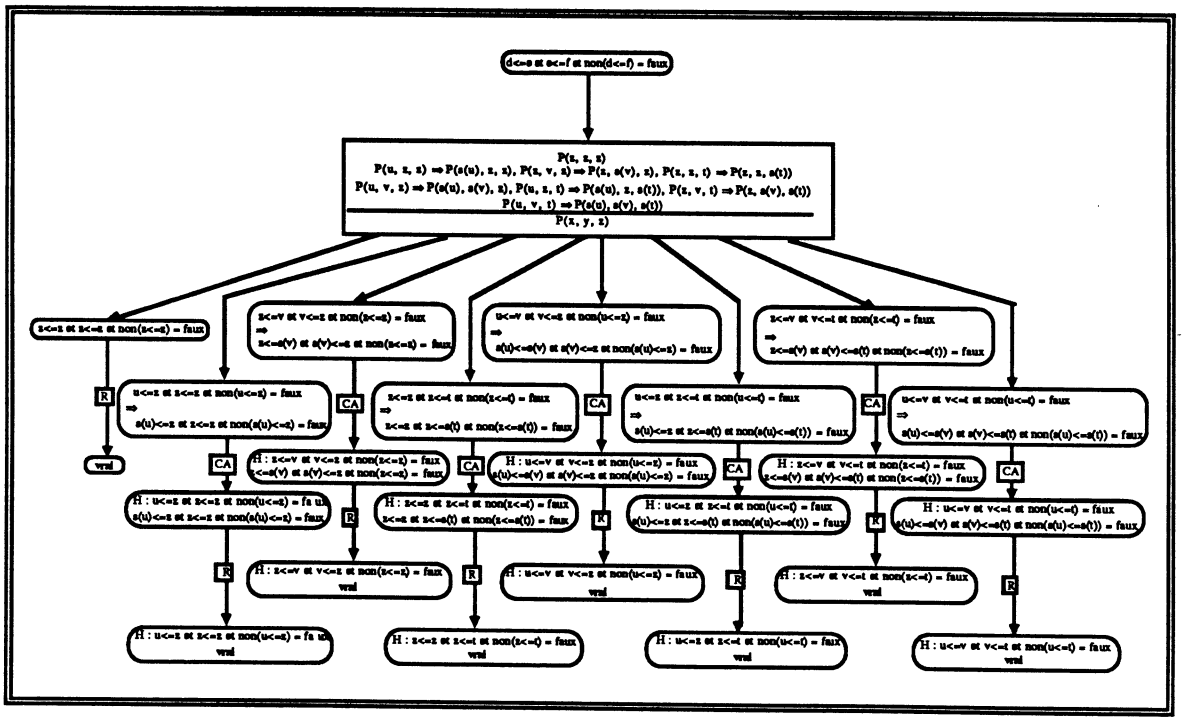
Le plus grand nombre de sous-buts à résoudre pour une démonstration lors de cette session de validation a été huit. Ces démonstrations concernent les théorèmes numéro 11 et 17 relatifs à la transitivité de relations.

Les deux figures suivantes ont pour but de mettre en évidence une dernière fois l'analogie pouvant exister entre les démonstrations de deux théorèmes différents. On peut se rendre compte, de plus, de la complexité de telles démonstrations[1], ce qui incite d'autant plus à tenter de réutiliser des schémas de démonstrations déjà réalisées.

[1]La loupe n'est pas fournie avec ce document...



Arbre de preuve de (d='c et c='f et non(d='f') = faux)



Arbre de preuve de (d<=e et e<=f et non(d<=f) = faux)

A l'aide des figures précédentes, on ne peut que remarquer la ressemblance des démonstrations réalisées. A quelques exceptions près (cas où il a été nécessaire

d'utiliser des règles de réécritures non orientables), les démonstrations se résument à une décomposition en sous-buts par un schéma d'induction suivie de réduction par réécriture des différents sous-buts obtenus.

On aimerait alors, et ceci est encore plus évident lorsqu'on réalise pratiquement les démonstrations, avoir la possibilité de dire *pour démontrer ce théorème, je veux essayer d'utiliser tel squelette de démonstration.* Eventuellement, sur tel sous-but ainsi obtenu (l'application d'un squelette peut ne pas suffire pour démontrer le théorème initial, c'est le cas des théorèmes numéro 8 et 13), il faut avoir la possibilité de dire *je veux appliquer tel autre squelette de démonstration.* Finalement, si la démonstration réussit, on peut considérer un nouveau squelette de démonstration, résultat d'une certaine composition de squelettes déjà existant.

Une démonstration n'est alors plus considérée comme un simple processus servant à valider des définitions, mais comme un terme particulier d'une algèbre. On peut ainsi construire des preuves élémentaires, composer entre elles ces preuves élémentaires pour obtenir des schémas plus élaborés et éventuellement raisonner sur ces termes de preuve. C'est cette idée d'algèbre de preuve qui est soulevée dans ce chapitre.

7.3- Manipulation des preuves en LCF

Voyons maintenant comment sont considérées les preuves dans le système LCF.

Nous avons vu dans le chapitre précédent que le système LCF utilise les notions de *tactique*, de *validation*[1], de *tacticielle* et de *stratégie*.

Les tactiques permettent d'obtenir une liste de sous-buts à partir d'un but. Les tacticielles sont des combinateurs de tactiques. Les stratégies décrivent un enchaînement de tacticielles. Enfin, une validation permet de générer un nouveau théorème dans le système.

LCF n'introduit pas la notion d'algèbre de preuve. On voit toutefois qu'une preuve est représentée par une stratégie qui peut être manipulée comme un objet particulier (application, combinaison). De plus, on trouve les types prédéfinis *goal*, *proof*, *tactic* et *thm*. Le type *goal* correspond aux phrases du langage que l'on veut essayer de démontrer. Le type *proof* correspond à la fonction de validation retournée par l'application d'une tactique. Le type *thm* regroupe des phrases du langage qui ont été démontrées et qui sont donc des théorèmes. Enfin, le type *tactic* regroupe les tactiques que l'on peut appliquer au cours d'une preuve. On peut voir dans [GMW 79] des exemples de démonstrations et de manipulations de tactiques.

On peut ainsi dire que LCF utilise, sans la formaliser explicitement, la notion d'algèbre de preuves. Elle permet, dans ce cadre, de réaliser des démonstrations rigoureuses (seules les applications de tactiques, éventuellement combinées entre

[1]Il ne faut pas confondre cette notion appelée dans LCF validation, qui permet de générer des théorèmes après avoir utilisé des stratégies, et celle introduite dans cette thèse qui concerne certaines vérifications sémantiques relatives aux spécifications.

elles, peuvent générer des théorèmes) et de faciliter dans la plupart des cas les démonstrations (réutilisation de stratégies précédemment écrites).

7.4- Proposition pour LPG

7.4.1- Introduction

Nous allons proposer dans ce chapitre les grands traits des spécifications de différentes fonctionnalités d'un atelier de démonstration. Le travail présenté ici n'est qu'une première étape, et probablement pas la plus intéressante, dans l'étude de la notion d'algèbre de preuves.

Une prochaine étape consisterait, par exemple, à spécifier précisément le type abstrait des preuves en précisant comment les termes de cette algèbre sont construits et surtout en spécifiant différents opérateurs sur ces termes. On peut consulter par exemple [BC 87] qui parle essentiellement de ce type de problème.

La motivation principale de cette étude est d'essayer d'automatiser au maximum le déroulement d'une démonstration. Cela permettrait, appliqué au langage LPG, de faciliter les différentes validations sémantiques que le système peut proposer.

Pour ce faire, nous allons employer le formalisme du langage LPG et tenter de spécifier différentes fonctionnalités que l'on peut souhaiter trouver dans un atelier de démonstration. Pour des raisons de clarté, nous ne donnerons ici que la signature de chacune des spécifications.

Il est toutefois intéressant de signaler que ces différentes spécifications ont été écrites sur le papier en utilisant le langage LPG. L'utilisation de LPG pour réaliser ce travail est donc tout à fait envisageable en supposant tout de même l'existence de certaines fonctions non encore disponibles permettant, par exemple, d'accéder aux représentations internes des termes LPG eux-mêmes.

Les différentes fonctionnalités spécifiées ici sont fortement inspirées de celles présentes dans les systèmes OASIS et LCF, en les améliorant le cas échéant.

Ce travail pourrait débouché sur une implémentation d'un atelier de démonstration utilisable dans l'atelier de spécification LPG. La connexion existant actuellement avec l'outil OASIS ne serait alors plus indispensable. Le maniement des démonstrations en serait alors facilité, en considérant, par exemple, la réutilisation de schéma de démonstrations. De plus le système global deviendrait homogène dans le sens où un seul langage de programmation pour l'implémentation serait employé. Actuellement, le système est écrit en utilisant trois langages différents (PL1, PASCAL et PROLOG), ce qui complique sérieusement la portabilité et la maintenance du système.

Nous présentons tout d'abord le cadre du travail. On donne ensuite la liste des sortes qui sont manipulées ainsi que les principales fonctionnalités. On décrit alors un schéma de l'atelier de démonstration permettant de fixer les idées vues auparavant. On introduit diverses autres fonctionnalités. On présente enfin brièvement comment, au niveau commande du démonstrateur, on peut manipuler les définitions données précédemment.

7.4.2- Hypothèses réalisées

Considérant une certaine unité LPG U, on peut lui faire correspondre une signature $\Sigma_U=(S_U,\Omega_U)$. Associée à cette signature, on considère la présentation $P_U=(\Sigma_U,E_U)$. E_U est l'ensemble des axiomes universellement quantifiés notés $t_1==t_2$ ou $t_1==>t_2$ ainsi que des théorèmes préalablement démontrés notés $t_1==t_2$. Les symboles t_1 et t_2 sont des termes, contenant éventuellement des variables, construits à partir de la signature Σ_U.

Le symbole == indique que l'axiome ne peut pas être vu comme une règle de réécriture normale (on n'a pas de raison de considérer la règle $t_1 \rightarrow t_2$ plutôt que la règle $t_2 \rightarrow t_1$). Le symbole ==> indique par contre que l'on peut considérer la règle de réécriture $t_1 \rightarrow t_2$.

Partant de ces hypothèses, on aimerait créer la théorie associée à cette présentation. Autrement dit, on cherche à obtenir la liste de tous les théorèmes provenant des axiomes et des théorèmes déjà démontrés. Ce but est évident exagéré puisque cette liste est généralement infinie.

Pour ce faire, il nous faudra construire un système de réécriture $\Re=(\overline{R},\overline{E})$ où $\overline{R}$ représente une liste de règles de réécritures orientables et $\overline{E}$ représente une liste d'équations, non orientables qui pourront toutefois être utilisées pendant le processus de réécriture en prenant des précautions particulières.

7.4.3- Sortes manipulées

Nous allons commencer par donner les différentes sortes, ainsi que leurs constructeurs, qui seront manipulées dans l'atelier de démonstration.

Nous venons d'introduire la notion de système de réécriture. La première sorte à définir est donc syst_reecr

- c_syst_reecr : (syst_orient, syst_non_orient)→syst_reecr

qui est un produit cartésien constitué d'une part d'un système de réécriture orienté $\overline{R}$ (syst_orient) et d'un système non orienté $\overline{E}$ (syst_non_orient), d'autre part. On peut récupérer le système orienté et le système non orienté d'un système de réécriture par deux opérations d'extraction.

Ces deux dernières sortes sont définies de la manière suivante :

- c_syst_orient_vide : →syst_orient,
- ins_syst_orient : (regle,syst_orient)→syst_orient,
- c_syst_non_orient_vide : →syst_non_orient,
- ins_syst_non_orient : (equation,syst_non_orient)→syst_non_orient.

Chacun des systèmes peut être vu en quelque sorte comme une séquence d'objets. Le système orienté contient des règles de réécritures, le système non orienté contient des équations.

Les règles et les équations sont des produits cartésiens de termes :

- --> : (terme, terme)→regle,
- <--> : (terme, terme)→equation.

Avec ces deux constructeurs sont définis des opérateurs pour extraire les membres gauches et membres droits d'une règle ou d'une équation.

La sorte des termes regroupera les *constantes* et les *opérateurs* du langage, les *variables* ainsi que les expressions plus élaborées telles l'expression de liaison *soit* et l'expression conditionnelle *si*. Chacun de ces objets sera représenté par un produit cartésien :

- c_variable : (idf, sorte)→terme,
- c_constante : (idf, sorte)→terme,
- c_operateur : (idf, sorte, param)→terme,
- c_si : (cond, terme, terme)→terme,
- c_soit : (liaison, terme)→terme.

Les constantes sont définies à partir de leur identificateur *idf* et de leur sorte *sorte* qui sont, en fait, des chaînes de caractères. Les opérateurs sont définis de la même manière que les constantes en mentionnant leur paramètre *param* qui est donc une nouvelle sorte pouvant être représentée par un produit cartésien de termes dont le nombre d'éléments n'est pas fixé. L'expression conditionnelle *si* est construite à partir de trois termes. Le premier est particulier dans le sens où il doit être de la sorte des booléens. L'expression de liaison permet de nommer des expressions par l'intermédiaire de variables et de les utiliser dans un terme. On introduit de ce fait la sorte *liaison* qui lie une (des) variable (s) à un (des) terme (s).

- c_param : p_cart(terme)→param,
- c_liaison : seq[(terme, terme)]→liaison.

Une liaison est une séquence de produit cartésien de termes. Il faut remarquer que le premier terme de chacun de ces produits cartésiens doit être une variable.

Un système de réécriture dans notre cas est obtenu à partir d'un ensemble d'objets appelés *lpg_theoreme*, cet ensemble lui-même, que l'on peut voir comme étant une séquence, est appelé *lpg_theorie* :

- c_theorie_vide : →lpg_theorie,
- ins_theorie : (lpg_theoreme,lpg_theorie)→lpg_theorie,
- c_theoreme : (justification, terme, terme)→lpg_theoreme,
- c_axio_lpg : axiome_lpg→lpg_theoreme
- c_theo_lpg : theoreme_lpg→lpg_theoreme.

Un *lpg_theoreme* est composé d'une justification et de deux termes. La justification explique comment les deux termes ont été démontrés égaux. Un *lpg_theoreme* peut aussi être construit en prenant soit les axiomes présents dans le spécifications LPG (*c_axio_lpg*), soit en prenant les théorèmes qui ont été démontrés dans ces spécifications (*c_theo_lpg*).

On doit bien sûr avoir un opérateur pour extraire la justification d'un *lpg_theoreme*. Appliqué à l'un des deux constructeurs *c_axio_lpg* ou *c_theo_lpg*, cet opérateur rend la constante *est_theoreme* de la sorte des justifications. On a également la possibilité d'extraire le terme gauche ou le terme droit d'un *lpg_theoreme*.

La sorte des justifications, qui vient d'être introduite, regroupe les différentes manières de réaliser des preuves. On a donc un constructeur de justification pour chaque opérateur de démonstration fourni dans l'atelier :

- est_theoreme : →justification,
- reduction : justification→justification,
- etape_de_reduction : (regle,justification)→justification,
- specialisation : (terme,terme,justification)→justification,
- remplacer : justification→justification,
- etape_de_remplacement_gauche : (equation,justification)→justification,
- etape_de_remplacement_droit : (equation,justification)→justification,
- cas_automatique : seq[justification]→justification,
- cas_manuel : (seq[(terme,terme)],seq[justification])→justification,
- induction : (schema,seq[terme],seq[justification])→justification.

L'ensemble de *lpg_theoreme* appelé *lpg_theorie* est obtenu par la prise en compte de nouveaux théorèmes démontrés et, lors de l'étape d'initialisation, par les axiomes et théorèmes définis dans les spécifications LPG. D'où les deux sortes suivantes définies comme des couples d'entiers :

- c_axiome_lpg : (terme, terme)→axiome_lpg,
- c_theoreme_lpg : (terme, terme)→theoreme_lpg.

Les *axiome_lpg* ou *theoreme_lpg* sont des égalités entre termes données explicitement dans les spécifications LPG.

La raison d'être de cet atelier est de démontrer des théorèmes encore appelés *but*. Nous allons donc manipuler cette sorte définie comme étant un produit cartésien constitué d'hypothèses et d'une formule.

- c_but : (hypothese, formule)→but.

D'autres opérateurs peuvent extraire soit la partie hypothèse, soit la formule d'un but.

Les hypothèses peuvent être vues comme un système de réécriture. Chaque hypothèse est en fait une règle de réécriture ou une équation dont la priorité est plus élevée que les autres règles de réécriture ou équations. Cette notion de priorité intervient dans le choix des règles (équations) effectué par le système de réécriture. En présence d'hypothèses, le système essaie tout d'abord d'appliquer les règles définies dans les hypothèses avant de tenter d'appliquer les autres règles du système de réécriture.

- c_hypothese : (syst_orient,syst_non_orient)→hypothese.

De même que pour un système de réécriture, on peut extraire le système orienté ou le système non orienté d'une hypothèse.

Les formules sont des objets construits de différentes manières. Une formule peut être une égalité entre termes. Mais elle peut prendre aussi la forme d'une implication, d'une conjonction ou d'une disjonction entre deux formules ou peut être enfin la négation d'une formule. *vrai* et *faux* sont également des formules :

- === : (terme,terme)→formule,
- ===> : (formule,formule)→formule,
- et : (formule,formule)→formule,
- ou : (formule,formule)→formule,
- non : formule→formule,
- vrai : →formule,
- faux : →formule.

Une autre sorte introduite dans cet atelier de démonstration est celle des schémas d'induction (règle d'inférence) :

- c_schema : (seq[composant], composant)→schema.

Un schéma d'induction est un produit cartésien constitué d'une séquence de composants (les conditions du schéma) et d'un composant (le conséquent). Un opérateur permet de sélectionner les conditions d'un schéma d'induction, un autre peut accéder à son conséquent.

Un composant peut prendre l'une des deux formes suivantes :

- P(seq[terme]) ou bien
- composant ===> composant.

En d'autres termes, un composant peut être une proposition ou encore une implication. D'où les deux constructeurs de composant :

- P : seq[terme]→composant,
- ===> : (composant,composant)→composant.

On trouve également la sorte des preuves qui est définie à peu près comme la sorte des justifications (même genre de constructeurs). Mais, à la différence des

justifications, les constructeurs de la sorte des preuves prennent en paramètre, non pas les objets paramètres des constructeurs de justification, mais des abstractions de ceux-ci :

- axiomatisation : →preuve,
- reduction : preuve→preuve,
- etape_de_reduction : (a_regle,preuve)→preuve,
- specialisation : (a_terme,a_terme,preuve)→preuve,
- remplacer : preuve→preuve,
- etape_de_remplacement_gauche : (a_equation,preuve)→preuve,
- etape_de_remplacement_droit : (a_equation,preuve)→preuve,
- cas_automatique : seq[preuve]→preuve,
- cas_manuel : (seq[(a_terme,a_terme)],seq[preuve])→preuve,
- induction : (a_schema,seq[a_terme],seq[preuve])→preuve.

On doit considérer ainsi l'abstraction d'une règle, d'une équation, l'abstraction d'un schéma d'induction, celle d'un composant, celle d'un terme et enfin celle d'une liaison.

Une abstraction de règle est la même sorte qu'une règle, mais les éléments sont construits à partie d'abstraction de terme. Il en va de même pour les abstractions d'équations :

- --> : (a_terme, a_terme)→a_regle,
- <--> : (a_terme, a_terme)→a_equation.

Une abstraction de schéma d'induction est construite à partir d'abstractions de composants :

- c_schema : (seq[a_composant], a_composant)→a_schema.

Une abstraction de composant, à son tour, est définie à partir d'abstractions de termes :

- P : seq[a_terme]→a_composant,
- ===> : (a_composant,a_composant)→a_composant.

Une abstraction de terme est définie à partir de divers constructeurs (du même genre que les constructeurs de terme) :

- c_variable : (idf, t)→a_terme,
- c_constante : (idf, t)→a_terme,
- c_operateur : (idf, t)→a_terme,
- c_si : (cond, a_terme, a_terme),→a_terme
- c_soit : (a_liaison, a_terme)→a_terme.

Mais il faut remarquer que la sorte des constantes, opérateurs et variables n'apparaît plus et est remplacée par le symbole *t*. De plus, on ne conserve pas les paramètres de l'opérateur. Ce symbole, *t*, et cette disparition des paramètres d'un opérateur constituent l'abstraction en elle-même dans la mesure où on oublie la sorte d'un objet particulier et on la généralise à des objets ayant une sorte quelconque *t*.

Vient enfin l'abstraction de liaison :

- c_liaison : seq[(a_terme, a_terme)]→a_liaison.

De même que pour la sorte des liaisons, il faut remarquer que le premier élément de chaque produit cartésien de la séquence définissant l'abstraction de liaison doit être une abstraction de variable.

7.4.4- Principales fonctionnalités

Les fonctionnalités que l'on souhaite avoir dans cet atelier de démonstration se répartissent en différents groupes.

On a tout d'abord des fonctions qui concernent la réécriture traditionnelle. On ne fait que réécrire un but soit en sa forme normale, par la fonction *reduction*, soit en utilisant une seule règle de réécriture, par la fonction *etape_de_reduction*, ce qui peut éventuellement conduire à sa forme normale.

- reduction : but→but
  reduction(b)==b' où *b'* est obtenu à partir de *b* et en utilisant $\overline{R}$: $b \xrightarrow{\overline{R}} b'$,
- etape_de_reduction : (regle,but)→but
  etape_de_reduction(g→d,b)==b' où *b'* est obtenu à partir de *b* en utilisant la règle *g→d* : $b \xrightarrow{g \to d} b'$.

Une fonctionnalité pouvant être utile est la spécialisation. Elle permet, étant donné un but, de substituer une variable particulière de ce but par un terme.

- specialisation : (terme,terme,but)→but
  specialisation(t,x,b)==b' où *b'* est le but *b* dans lequel la variables *x* est substituée par le terme *t* : b'=b[x←t].

D'autres fonctionnalités intéressantes sont les fonctionnalités concernant la réécriture permutative. Dans ce cas, des règles non orientables, comme celle définissant la commutativité d'un opérateur, sont utilisées conjointement avec les règles de réécriture traditionnelles.

Le problème que l'utilisation de telles règles peut poser est que le processus de réécriture ne termine pas. Considérant, par exemple, l'addition + sur les entiers et l'équation définissant la commutativité, on peut réécrire le terme x+y en y+x, qui peut se réécrire en x+y, etc... Il faut donc faire attention en utilisant de telles fonctionnalités en essayant de détecter, par exemple, d'éventuelles boucles [Paul 85]. On peut définir les fonctions suivantes :

• remplacer : but→but

remplacer(b)==b' où le but *b'* est obtenu à partir du but *b* et du système de réécriture permutatif $\mathfrak{R}$: $b \xrightarrow{\mathfrak{R}} b'$,

• etape_de_remplacement_gauche : (equation,but)→but

etape_de_remplacement_gauche(tg==td,b)==b' où on calcule le nouveau but *b'* en appliquant une seule fois l'équation *tg==td* dans le sens *tg→td* au but *b* : $b \xrightarrow{tg \to td} b'$,

• etape_de_remplacement_droit : (equation,but)→but

etape_de_remplacement_droit(tg==td,b)==b' où on calcule le nouveau but *b'* en appliquant une seule fois l'équation *tg==td* dans le sens *td→tg* au but *b* : $b \xrightarrow{td \to tg} b'$.

La fonction *remplacer* permet de réécrire un but en un autre but en considérant aussi bien les règles de réécritures orientables que non orientables. Les fonctions *etape_de_remplacement_gauche* et *etape_de_remplacement_droit* permettent d'utiliser une règle de réécriture non orientable respectivement dans le sens membre gauche vers membre droit ou bien dans le sens membre droit vers membre gauche, et ceci une seule fois.

On peut introduire des fonctions de décomposition par cas [Paul 85] qui permettent de décomposer un but en sous-buts selon l'aspect général du but initial.

• cas_automatique : but→seq[but].

La séquence du buts est obtenue après une analyse automatique de la forme du but en détectant, par exemple, la présence de conjonction, d'implication, etc.

• cas_manuel : (seq[(terme,terme)],but)→seq[but].

On analyse dans ce cas la séquence de couples de termes donnée de façon à couvrir tous les cas. Par exemple, l'utilisation suivante de *cas_manuel*

cas_manuel ([(x,0)],b)

va donner la séquence [b_1, b_2] où b_1 est le but b sous l'hypothèse x=0→vrai et b_2 est le but b sous l'hypothèse x=0→faux.

Il faut bien sûr que les deux termes de chaque couple de la séquence soient de la même sorte et qu'une relation d'égalité sur cette sorte soit définie.

Pour compléter la décomposition de but en sous-buts, on introduit une fonction d'induction qui permet une telle décomposition en fonction d'un schéma d'induction.

• induction : (schema,seq[terme],but)→seq[but]

On doit imposer la restriction suivante. Les termes présents dans la séquence fournie en paramètre de cette fonction ne peuvent être que des variables qui figurent dans le but.

L'application de cette fonction à un but, en utilisant un schéma d'induction et une séquence de variables, produit une séquence de sous-buts. Il y a autant de sous-buts qu'il y a de composant dans la séquence de composants du schéma d'induction. Chaque sous-but est obtenu en instanciant son composant correspondant par substitution de la variable sur laquelle on fait l'induction par le terme proposé par le composant.

Si on considère par exemple le schéma d'induction sur les naturels :

$$\frac{P(0), P(y) \Rightarrow P(succ(y))}{P(x)}$$

et le but 0+n==n, on aura

```
induction ( c_schema ([P(0), P(y)===>P(succ(y))], P(x)),
            [n],
            c_but (  c_hypothese (c_syst_orient_vide, c_syst_non_orient_vide),
                     0+n===n))
==
  [  c_but (  c_hypothese (c_syst_orient_vide, c_syst_non_orient_vide),
              0+0===0),
     c_but (  c_hypothese (c_syst_orient_vide, c_syst_non_orient_vide),
              (0+y === y) ===> (0+succ(y) === succ(y)))].
```

Dans cet exemple, l'opérateur *c_schema* construit un schéma d'induction, *c_but* construit un but, *c_hypothese* construit une hypothèse, *c_syst_orient_vide* et *c_syst_non_orient_vide* construisent respectivement un système de réécriture vide et un système de réécriture non orientable vide.

Autrement dit, l'application de ce schéma d'induction sur ce but en utilisant la variable *n* produit les deux sous-buts :

- 0+0===0 et
- (0+y===y) ===> (0+succ(y)===succ(y)).

7.4.5- Schéma de l'atelier de démonstration

On peut avoir une vue générale de l'atelier de démonstration en regardant la figure suivante. Les flèches ⟶ indiquent les différentes fonctionnalités envisagées. Les flèches ⟶ représentent un transfert d'information (les axiome_lpg sont par exemple récupérés et inclus dans la lpg_theorie). Les flèches ⟶ signalent également un transfert d'information. Mais dans ce dernier cas, l'information est transformée. Il faut calculer en effet les objets de la sorte preuve à partir des objets de la sorte lpg_theorie. Le système de réécriture doit être obtenu de manière aussi automatique que possible à partir de la lpg_theorie.

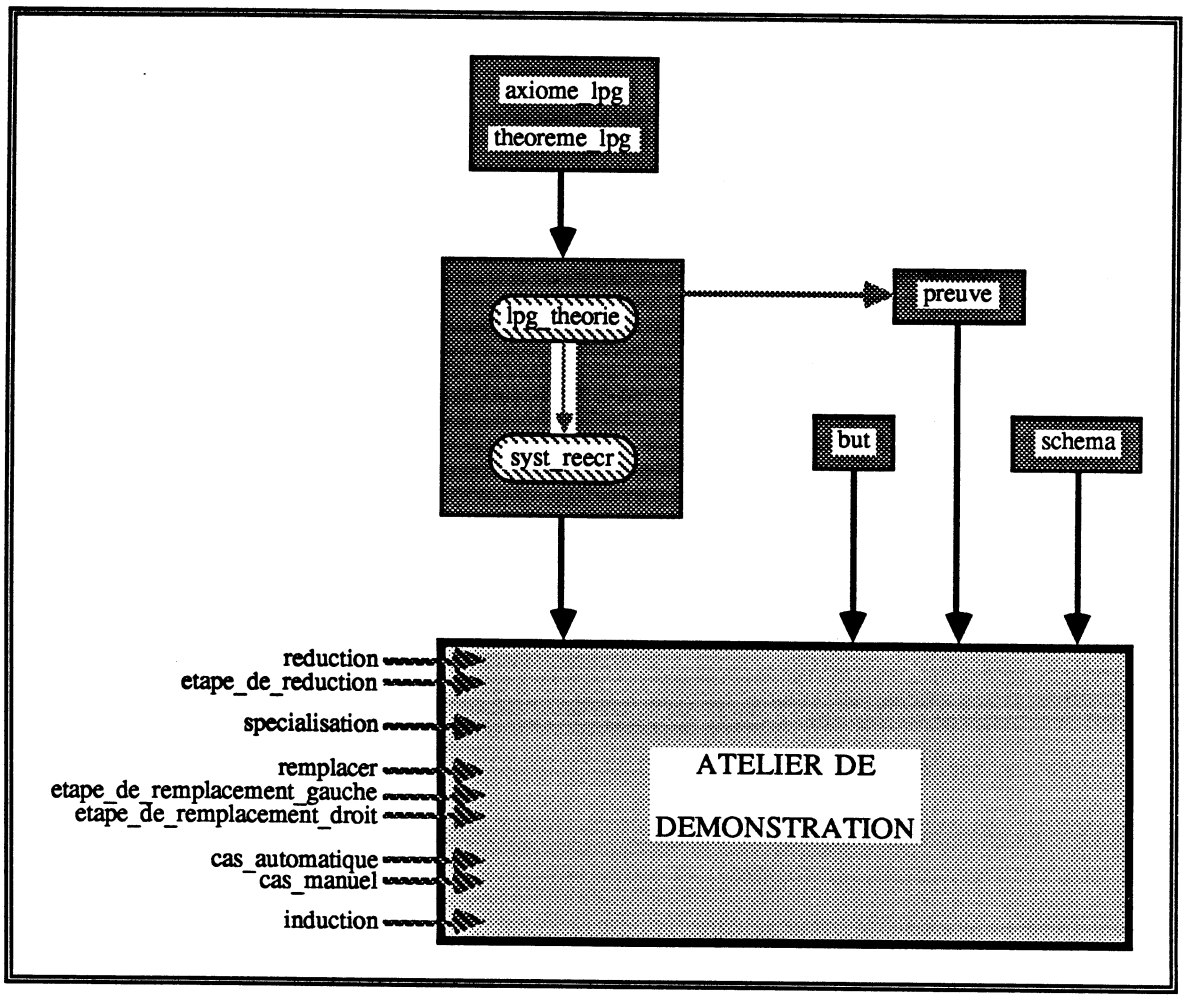


Schéma de l'atelier de démonstration

La partie principale de l'atelier contient les spécifications de ses différentes fonctionnalités ainsi que d'autres qui seront présentées ultérieurement.

Cet atelier prend ses informations à partir du système de réécriture, d'un but et de schémas d'induction.

Les buts peuvent être donnés par l'utilisateur en utilisant les constructeurs de cette sorte. Il serait également possible d'alimenter automatiquement cet atelier avec des théorèmes à démontrer calculés à partir des spécifications contenues dans le système LPG [BD 87], [Dra 88].

Les schémas d'induction peuvent être également donnés par l'utilisateur. Une autre solution consiste à les calculer automatiquement en considérant les constructeurs de la sorte sur laquelle on travaille. Prenons comme exemples les booléens, les naturels et les séquences.

Les constructeurs traditionnels des booléens sont :

- vrai : -> bool et
- faux : -> bool.

On peut en déduire le schéma d'induction

$$\frac{P(vrai),\ P(faux)}{P(b)}$$

dans lequel on ne fait qu'examiner les constantes de la sorte.

Les nombre naturels sont construits de la manière suivante :

- 0 : -> nat,
- succ : nat -> nat.

Ce qui peut donner le schéma d'induction

$$\frac{P(0),\ P(y) \Rightarrow P(succ(y))}{P(x)}.$$

On considère à nouveau la constante de la sorte et on exprime la façon suivant laquelle les autres valeurs de la sorte sont obtenues en utilisant l'implication logique.

Prenons enfin l'exemple des séquences pour lesquelles les constructeurs sont :

- nil : -> seq[elem] et
- <+ : (elem, seq[elem]) -> seq[elem].

On peut en déduire le schéma d'induction :

$$\frac{P(nil),\ P(s') \Rightarrow P(e<+s')}{P(s)}$$

avec un raisonnement analogue à celui utilisé pour les naturels.

L'induction vue précédemment est réalisée sur une variable. On peut également générer des schémas d'induction sur plusieurs variables. En reprenant l'exemple des entiers naturels, on peut construire le schéma d'induction sur deux variables

$$\frac{P(0,0),\ P(u,0) \Rightarrow P(succ(u),0),\ P(0,v) \Rightarrow P(0,succ(v)),\ P(u,v) \Rightarrow P(succ(u),succ(v))}{P(x,y)}.$$

Ce schéma est obtenu en considérant les variables d'induction les unes après les autres individuellement et en fixant les autres variables aux valeurs des constantes de la sorte. Cela revient ainsi à faire de l'induction sur une variable. On obtient ainsi les cas :

$$\left.\begin{array}{l} \bullet\ P(0,0) \\ \bullet\ P(u,0) \Rightarrow P(succ(u),0) \end{array}\right\} \text{induction sur la première variable}$$

• P(0,v)⇒P(0,succ(v)) induction sur la deuxième variable, son cas trivial, P(0,0), est pris en compte lors de l'induction sur la première variable.

On considère ensuite les couples de variables. On réalise alors de l'induction sur les deux variables en même temps. Sur notre exemple, cela permet d'obtenir le cas :

• P(u,v)⇒P(succ(u),succ(v))

Son cas trivial est déjà pris en compte lors de l'induction sur la première variable.

Pour trois variables, comme on peut le voir dans [Dra 88], on fixe le cas trivial, puis on réalise l'induction sur chacune des variables prise séparément. On réalise ensuite l'induction sur les couples de variables. On fait enfin l'induction sur les trois variables en même temps. On obtient, dans le cas d'un schéma d'induction sur les naturels, une décomposition en huit sous-buts.

7.4.6- Autres fonctionnalités

D'autres fonctionnalités doivent être spécifiées pour permettre la définition des fonctions principales que nous venons de voir. Les fonctions principales travaillent sur des buts. Maintenant, il faut s'intéresser à la réécriture des formules et des termes.

On trouve ainsi des fonctions concernant la réécriture normale :

• reduction_f : formule→formule
reduction_f(f)==f' avec *f'* qui est obtenu à partir de *f* en utilisant $\overline{R}$: $f \xrightarrow{\overline{R}} f'$,

• etape_de_reduction_f : (regle,formule)→formule
etape_de_reduction_f(g→d,f)==f' où on applique la règle *g→d* sur *f* pour obtenir *f'* : $f \xrightarrow{g \to d} f'$

• reecrire : terme→terme
reecrire(t)==t' où *t'* est obtenu à partir de *t* en utilisant le système de réécriture $\overline{R}$: $t \xrightarrow{\overline{R}} t'$,

• etape_de_reecrire : (regle,terme)→terme
etape_de_reecrire(g→d,t)=t' ce qui veut dire que l'on applique la règle de réécriture *g→d* au terme *t* pour calculer le nouveau terme *t'* : $t \xrightarrow{g \to d} t'$.

La fonction *reduction_f* permet de réécrire une formule en utilisant un système de réécriture. L'utilisation d'une seule règle pour réécrire une formule est possible grâce à la fonction *etape_de_reduction_f*.

On trouve également la fonction *reecrire* qui permet de réécrire un terme en utilisant un système de réécriture ainsi que la fonction *etape_de_reecrire* qui permet d'effectuer un seul pas de réécriture sur un terme.

On a également d'autres fonctions concernant la réécriture permutative.

- remplacer_f : formule→formule
  remplacer(f)==f' où on applique le système de réécriture $\mathfrak{R}$ sur la formule *f* pour calculer la nouvelle formule *f'* : $f \xrightarrow{\mathfrak{R}} f'$
- etape_de_remplacement_gauche_f : (equation,formule)→formule
  etape_de_remplacement_gauche_f(tg==td,f)==f' où on considère l'équation *tg==td* comme la règle de réécriture orientée à priori suivant *tg→td* pour calculer *f'* : $f \xrightarrow{tg \to td} f'$
- etape_de_remplacement_droit_f : (equation,formule)→formule
  etape_de_remplacement_droit_f(tg==td,f)==f' où on considère l'équation *tg==td* comme la règle de réécriture orientée cette fois ci suivant *td→tg* pour calculer *f'* : $f \xrightarrow{td \to tg} f'$
- reecrire_perm : terme→terme
  reecrire_perm(t)==t'. De même que pour la réécriture permutative de but, on obtient *t'* en utilisant le système de réécriture permutatif $\mathfrak{R}$ et en partant du terme *t* : $t \xrightarrow{\mathfrak{R}} t'$.
- etape_de_reecrire_perm_gauche : (equation,terme)→terme
  etape_de_reecrire_perm_gauche(tg==td,t)==t' où *t'* s'obtient à partir de *t* en utilisant la règle *tg→td* : $t \xrightarrow{tg \to td} t'$.
- etape_de_reecrire_perm_droit : (equation,terme)→terme
  etape_de_reecrire_perm_droit(tg==td,t)==t' où *t'* est obtenu en réécrivant *t* avec la règle *td→tg* : $t \xrightarrow{td \to tg} t'$.

La fonction *remplacer_f* permet de réécrire une formule en utilisant un système de réécriture comportant éventuellement des règles non orientables. On trouve bien sûr les fonctions *etape_de_remplacement_gauche_f* et *etape_de_remplacement_droit_f* qui réalise un seul pas de réécriture sur une formule en utilisant un système de réécriture ayant éventuellement des règles non orientables.

De la même manière, on trouve les fonctions équivalentes sur les termes *reecrire_perm*, *etape_de_reecrire_perm_gauche* et *etape_de_reecrire_perm_droit.*

Le mécanisme de réécriture d'un terme *t* implique un certain parcours des occurences de ce terme. Plusieurs parcours sont possibles, comme par exemple *en profondeur d'abord et en commençant par la gauche.* Le choix d'un parcours s'appelle une stratégie.

On peut définir diverses fonctions qui étant donné un terme délivrent une séquence de toutes ses occurences. Cette séquence définit donc un parcours ou une

stratégie possible pour la réécriture. Il serait alors intéressant de paramétrer le système de réécriture par une fonction générique définissant une stratégie sous sa forme la plus générale (une séquence d'occurences) et, lors de l'utilisation effective de ce système de réécriture, de donner une stratégie bien précise.

Viennent enfin diverses autres fonctionnalités plus ou moins évidentes.

On doit être capable de pouvoir substituer une variable d'une formule ou d'un terme par un autre terme :

- specialisation_f : (terme,terme,formule)→formule
  specialisation_f(t,x,f)==f' où *f'* est calculée à partir de *f* dans laquelle les occurences de la variables *x* sont remplacées par le terme *t* : f'==f[x←t],
- substituer : (terme,terme,terme)→terme
  substituer(t',x,t)==t'' où *t''* s'obtient en substituant les occurences de la variable *x* de *t* par *t'* : t''==t[x ←t'].

La substitution sur une formule est réalisée par la fonction *specialisation_f*, celle relative aux termes est faite par la fonction *substituer*.

Une fonction particulière doit permettre de calculer un système de réécriture à partir d'un objet de la sorte *lpg_theorie*.

- faire_syst_reecr : lpg_theorie→syst_reecr.

Diverses autres fonctions doivent être définies pour spécifier toutes les précédentes. On a, par exemple, *union_hypothese* qui permet de réaliser une union ensembliste entre deux objets de sorte *hypothese*. Ce qui implique la définition de l'union de deux systèmes orientés et l'union de deux systèmes non orientés.

Une fonction *orientable?* doit indiquer si étant donné une équation, on peut l'orienter en une règle de réécriture ou non. Etant donné une séquence de couples de termes, on doit être capable également de construire une séquence d'objets de sorte *hypothese*. On peut être amené à insérer une équation ou une règle de réécriture dans un système de réécriture.

Une autre fonction doit permettre de déduire un sous-but à partir d'un composant (issu d'un schéma d'induction), d'une séquence de variables et d'un but initial.

Afin de faciliter la manipulation de certains objets, tels que les schémas d'induction, les preuves, les règles de réécriture, il serait intéressant de pouvoir les nommer, comme c'est le cas dans le système DEVA [SWGC 88]. Un certain nombre de fonctions doivent ainsi être définies permettant d'associer un nom, représenté par exemple par une chaîne de caractères, à chacun de ces objets.

7.4.7- Commandes du démonstrateur

Il est nécessaire maintenant de définir diverses fonctions permettant de manipuler l'ensemble des définitions précédentes.

Il faut tout d'abord définir une fonction qui construit un terme de la sorte des preuves étant donné un objet de sorte *theoreme_lpg*.

- extrait_preuve : lpg_theoreme→preuve

Cette fonction analyse la justification associée à un *lpg_theoreme* et en extrait un terme de la sorte des preuves.

Il faut également définir les diverses fonctions globales de démonstration de l'atelier. Ces fonctions sont construites sur les fonctions déjà définies de démonstration telles que *reduction*, *specialisation*, *induction*, et doivent gèrer les termes de la sorte *lpg_theoreme* en construisant automatiquement la justification associée à chaque *lpg_theoreme*.

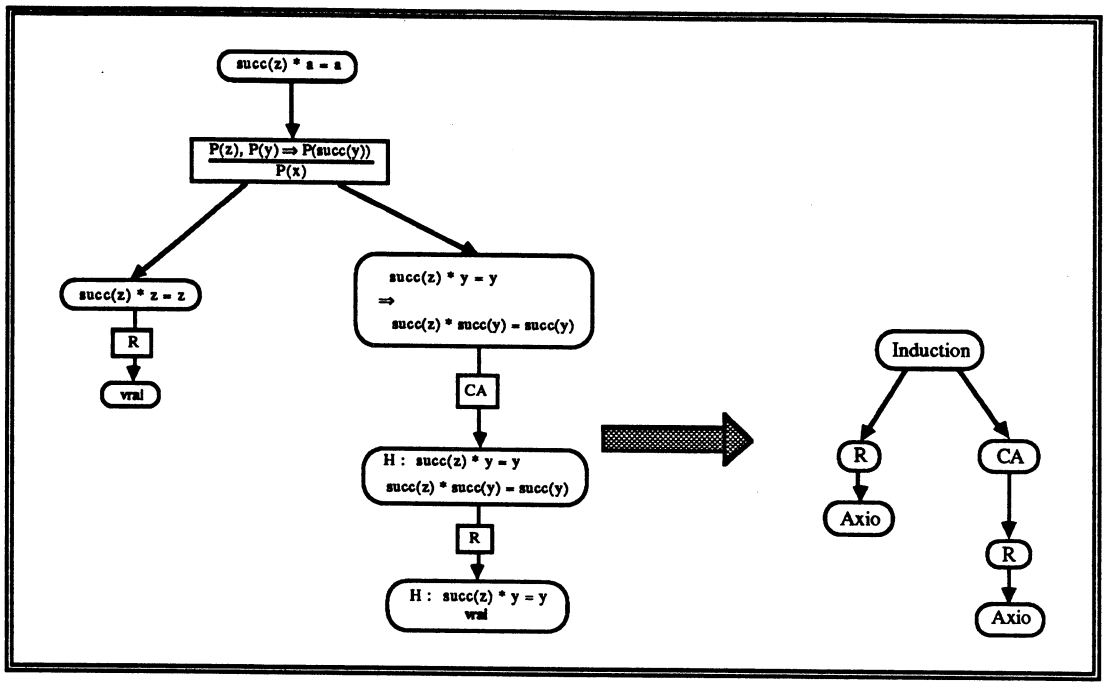
Il faut enfin définir une fonction particulière qui prenne en compte les objets de la sorte des preuves. Cette fonction prend en paramètre un terme de la sorte *preuve* et un *but* et retourne un *lpg_theoreme*.

- demontrer : (preuve, but)→lpg_theoreme

Cette fonction doit essayer de suivre les indications présentes dans le terme de preuve en appliquant les différentes étapes au théorème courant que l'on veut démontrer. La spécification de cette fonction doit faire en sorte de réduire au maximum les interventions de l'utilisateur.

7.4.8- Exemple d'extraction de preuve

Reprenons pour cet exemple le schéma de la démonstration du théorème sur les naturels $\forall x \in nat,\ 1*x=x$.



Extraction de la justification d'une démonstration

De ce schéma de démonstration, on peut extraire, comme on peut le voir dans la partie gauche de la figure précédente, le squelette de la démonstration, à savoir, les différentes étapes qui ont permis de réaliser cette démonstration. Ce squelette peut se représenter graphiquement sous la forme d'une arborescence. C'est un terme de la sorte *justification*. On peut le noter :

```
induction(
  c_schema(
     [P([c_constante(0,nat)]),
     ===>(  P([c_variable(y,nat)]),
            P([c_operateur(succ,nat,c_param(p_cart(c_variable(y,nat))))])
         )
     ],
     P([c_variable(x,nat)])),
  [c_variable(x,nat)],
  [  reduction(est_theoreme),
     cas_automatique([reduction(est_theoreme)])
  ])
```

Le terme de la sorte *preuve* issu de cette justification est à peu près identique. Il est schématisé dans la partie droite de la précédente figure. Il représente en quelque sorte les principales étapes de la démonstration sans trop descendre dans le détail.

Sur notre exemple, on mémorise le fait que l'induction a été utilisée. Puis nous avons employé d'une part la réduction qui conduit à un axiome et d'autre part une analyse automatique de cas suivie d'une réduction qui doit mener également à un axiome. Sur la figure, il manque quelques renseignements comme, par exemple, quel type de schéma d'induction a été utilisé.

Cette arborescence est donc un terme de la sorte des preuves que l'on peut noter de la manière suivante :

```
p == induction(
        c_schema(
           [P([c_constante(0,t)]),
           ===>(  P([c_variable(y,t)]),
                  P([c_operateur(succ,t)])
                )
           ],
           P([c_variable(x,t)])),
        [c_variable(x,t)],
        [  reduction(axiomatisation),
           cas_automatique([reduction(axiomatisation)])
        ])
```

Les mots écrits en caractères **gras** soulignent les différences la preuve et la justification. Le terme lui-même est nommé *p*.

Ce terme de preuve étant mis à la disposition de l'utilisateur dans l'atelier de démonstration, on peut essayer de l'utiliser pour conduire une autre démonstration.

Supposons que nous ayons la spécification suivante des séquences.

```
type Seq
requires Formal_Sort [elem]
sorts
   seq
constructors
   nil: -> seq[elem]
   <+: (elem, seq[elem]) -> seq[elem]
operators
  reverse: seq[elem] -> seq[elem]
  append: (seq[elem], seq[elem]) -> seq[elem]
variables
  e: elem
  s,s1,s2: seq[elem]
equations
  1: reverse(nil) ==> nil
  2: reverse(e<+s) ==> append(reverse(s), e<+nil)
  3: append(s1, nil) ==> s1
  4: append(nil, s2) ==> s2
  5: append(e<+s1, s2) ==> e<+append(s1, s2)
  6: append(append(s1, s2), s3) ==> append(s1, append(s2, s3))
theorems
  1: reverse(append(s1, s2)) == append(reverse(s2), reverse(s1))
end Sort
```

Supposons, par ailleurs, que le schéma d'induction suivant sur les séquences soit disponible dans l'atelier de démonstration.

$$S : \frac{P(nil),\ P(s') \Rightarrow P(e<+s')}{P(s)}$$

Afin de démontrer le théorème présent dans la rubrique *theorems* de cette spécification, on peut écrire *demontrer(p, b)* où *p* désigne le terme de preuve précédemment calculé et *b* repère le but fabriqué à partir du théorème de la spécification.

On demande d'utiliser la preuve *p* pour démontrer *b*. On retrouve dans *p* le fait qu'il est nécessaire d'utiliser l'induction sur une variable précise avec un schéma d'induction précis. On essaie alors de retrouver un schéma d'induction connu du système qui filtre l'abstraction présente dans *p*. On peut s'apercevoir que *S* peut convenir :

- t ← seq,
- 0 ← nil,
- y ← s',
- succ ← <+ et
- x ← s.

Il suffit, en effet, de considérer la sorte *seq* à la place de la sorte *t*. La constante *0* peut être substituée par la constante *nil*. L'opérateur *succ* est remplacé par *<+*.

La preuve nous dit qu'il faut faire l'induction sur une variable de sorte *t*, donc *seq*. On peut prendre la variable *s1* du théorème, par exemple. Toujours en suivant la preuve, le résultat de l'induction doit être deux sous-buts. L'application de *S* donne effectivement les sous-buts

- reverse(append(nil,s2)) == append(reverse(s2),reverse(nil)) et
  reverse(append(s',s2)) == append(reverse(s2),reverse(s'))
- ⇒
  reverse(append(e<+s',s2)) == append(reverse(s2),reverse(e<+s'))

Sur le premier sous-but, la preuve nos indique qu'il faut simplement faire de la réduction par réécriture. La réécriture du membre gauche donne[1] :

- reverse(append(nil,s2)) $\xrightarrow{3}$ reverse(s2)

Celle du membre droit donne successivement :

- append(reverse(s2),reverse(nil))
  $\xrightarrow{1}$ append(reverse(s2),nil)

[1]La réécriture est symbolisée ici par une flèche surmontée du numéro de la règle qui a été utilisée ou bien du caractère *H* dans le cas de l'utilisation de l'hypothèse d'induction.

- append(reverse(s2),nil)
  $\xrightarrow{4}$ reverse(s2)

La preuve nous dit qu'on peut obtenir un axiome. On a effectivement égalité entre le membre gauche et le membre droit.

De la même manière, on peut suivre les indications de la preuve pour démontrer le deuxième sous-but. On fait tout d'abord une analyse automatique de cas, ce qui donne le sous-but

reverse(append(e<+s',s2)) == append(reverse(s2),reverse(e<+s')),

sous l'hypothèse *H*

reverse(append(s',s2)) == append(reverse(s2),reverse(s')).

La preuve nous indique maintenant qu'il faut réécrire ce dernier sous-but. Le membre gauche se réécrit suucessivement

- reverse(append(e<+s',s2))
  $\xrightarrow{5}$ reverse(e<+append(s',s2))
- reverse(e<+append(s',s2))
  $\xrightarrow{2}$ append(reverse(append(s',s2)),e<+nil)
- append(reverse(append(s',s2)),e<+nil)
  $\xrightarrow{H}$ append(append(reverse(s2),reverse(s')),e<+nil)
- append(append(reverse(s2),reverse(s')),e<+nil)
  $\xrightarrow{6}$ append(reverse(s2),append(reverse(s'),e<+nil))

La réécriture du membre droit donne

- append(reverse(s2),reverse(e<+s'))
  $\xrightarrow{2}$ append(reverse(s2),append(reverse(s'),e<+nil))

On trouve alors l'égalité entre les membres gauche et droit. Le théorème est de ce fait démontré.

Il faut remarquer que cette démonstration a pu se dérouler sans l'intervention de l'utilisateur, simplement en suivant les indications de la preuve donnée en paramètre.

Il n'est bien sûr pas évident que la réécriture débouche sur l'égalité entre les membres gauche et droit. Dans un tel cas, le système rend la main à l'utilisateur qui peut trouver une solution pour débloquer la situation. Il peut ajouter par exemple un axiome à la spécification ou bien démontrer un théorème intermédiaire qui servira de lemme pour le théorème courant.

7.5- Vers une algèbre de preuves

Nous rappelons que le but de ce chapitre n'est que d'introduire la notion d'algèbre de preuves. En effet, ce travail ayant été commencé très tardivement au cours de cette thèse, nous n'avons pas eu le temps d'explorer ce domaine.

Nous avons décrit succintement un atelier de démonstration en utilisant le formalisme de LPG. La plupart des fonctions décrites ont été également spécifiées à l'aide de ce formalisme. Toutefois, ces spécifications n'ont pas été analysées par l'interpréteur LPG. Il est donc fort probable qu'elles contiennent des erreurs. Il serait néanmoins possible de franchir cette étape.

D'autres fonctions, pour être implémentées, nécessitent des primitives n'existant actuellement pas en LPG. C'est le cas des fonctions telles que *c_variable* qui ont besoin d'accéder aux représentations internes de LPG. Ces fonctions n'ont donc pas été spécifiées.

Enfin, à l'opposé, les fonctions de haut niveau telles que *demontrer* n'ont pas été spécifiées. C'est également le cas des fonctions qui manipulent les justifications ou les preuves. Comme principale conséquence, un certain niveau d'abstraction a été proposé pour calculer une preuve à partir d'une justification. Il est fort probable que l'on puisse rafiner ce niveau afin d'assouplir, par exemple, la réutilisation d'une preuve.

8- Conclusion

Le problème qui est à l'origine de cette thèse était de pouvoir prouver que des spécifications écrites de manière structurée en LPG vérifient certaines conditions sémantiques exprimées dans ce même langage. Ces différentes vérifications se traduisent finalement par des démonstrations de formules logiques.

Nous avons donc formalisé les différentes relations, telles que la paramétrisation, existant entres les unités LPG. Cette formalisation nous a conduit à exhiber les conditions qui doivent être vérifiées afin de pouvoir affirmer qu'une spécification donnée est correcte sémantiquement. Nous avons de ce fait formalisé également ces conditions sémantiques ainsi que le moyen de les retrouver, à partir des spécifications qu'un utilisateur peut écrire, et de les démontrer.

Afin d'implémenter le résultat de cette étude, il était nécessaire d'utiliser un démonstrateur de théorèmes pour pouvoir démontrer les formules logiques extraites des spécifications. Nous avons considéré le système OASIS qui offre un démonstrateur semi-automatique de théorèmes. On peut le qualifier de *semi-automatique* dans la mesure où il peut en fait assister l'utilisateur au cours d'une démonstration. Il permet essentiellement de réécrire des termes et de prendre en compte une certaine forme de règle d'inférence appelée schéma d'induction.

La partie implémentation de ce travail a été d'une part de réunir les deux systèmes LPG et OASIS et d'autre part de retrouver les différentes formules logiques à démontrer pour réaliser effectivement les validations sémantiques dans le cadre de LPG. Une importante partie de ce travail a donc consisté à étudier, d'une manière très détaillée, l'implémentation de ces deux systèmes. Après cette étude, nous les avons réunis par l'intermédiaire d'une interface. Cette réunion se traduit par la possibilité de transférer des informations directement entre les deux systèmes, c'est à dire sans passer par l'intermédiaire de fichiers. Les informations transférées peuvent être des équations dans le sens LPG-OASIS, ou bien *tel théorème a été effectivement démontré* dans le sens OASIS-LPG. D'autres informations peuvent également être échangées. C'est le cas, par exemple, des commandes de démonstrations intrinsèques à OASIS que l'on peut utiliser sous le système LPG. Le résultat de cette réunion permet ainsi de bénéficier des fonctionnalités de démonstrations offertes par OASIS directement et d'une manière transparente sous le système LPG.

Les essais de cette interface que nous avons réalisés ont mis en évidence l'avantage que l'on pourrait retirer de la réutilisation de schémas, ou de squelettes, de démonstration. Il n'est pas rare, en effet, que plusieurs démonstrations se ressemblent globalement. On peut trouver, par exemple, l'utilisation de schémas d'inductions similaires aux mêmes endroits de la preuve. Cette constatation est à l'origine de la notion d'algèbre de preuves qui est très brièvement abordée à la fin de cet ouvrage. Ce sujet de recherche mérite toutefois de s'y intéresser de plus près. Il peut déboucher effectivement sur la réutilisation de schémas de preuves mais également sur les manipulations de preuves en tant que termes d'une algèbre. On

peut imaginer par exemple de composer des preuves entre elles afin d'en créer d'autres.

On peut souligner trois points particuliers issus de ce travail.

Le premier est relatif à l'étude théorique des relations existant entre les différentes unités du système LPG. Le résultat est une formalisation de ces relations ainsi que des conditions sémantiques qui se trouvent de ce fait exhibées.

Le deuxième point concerne l'implémentation réalisée. Ce résultat pratique permet de réaliser effectivement des validations sémantiques de spécifications écrites en LPG. Il faut remarquer toutefois que cette implémentation a été réalisée sur une version relativement ancienne de LPG et sous un système d'exploitation, Multics, qui est appelé à disparaître d'ici peu de temps. Une nouvelle version de LPG existe actuellement sous le système UNIX, mais elle ne tient pas compte de cette notion de validation sémantique.

Enfin, le troisième point se rapporte à l'enseignement que l'on peut retirer de cette expérience. La démarche suivie pour réaliser cette implémentation peut être critiquée. Il aurait été probablement aussi simple d'implémenter un démonstrateur de théorèmes, offrant les mêmes fonctionnalités que celui de OASIS, spécialement adapté pour le système LPG plutôt que d'en utiliser un qui existe déjà.

La pertinence de ce résultat vient surtout du fait des différences d'implémentation de ces deux systèmes. Les structures de données sont totalement différentes (tables pour LPG, déclarations de fait pour OASIS). Les langages d'implémentation sont également totalement différents (PL1 pour LPG, PROLOG pour OASIS). Ces différences et l'absence d'interface pour manipuler les représentations internes des informations de OASIS ont nécessité une étude très détaillée des deux implémentations. Cette constatation est d'autant plus vraie pour un produit qui est en pleine évolution, ce qui est le cas de LPG.

La conséquence la plus importante est que la partie du système concernant les validations sémantiques n'a pas été reportée au cours du changement de version de LPG (de la version fonctionnant sous Multics à la version fonctionnant actuellement sous UNIX). L'effort serait en effet beaucoup trop important. Il faudrait non seulement traduire les programmes de l'interface, mais également transférer la totalité du système OASIS[1] et donc de disposer des différents langages dans lesquels sont écrits les systèmes.

Il ne faut toutefois pas renier systématiquement l'avantage de cette démarche. Le fait d'essayer de ne pas réinventer la roue garde généralement un côté attrayant. Dans notre cas, nous avions besoin d'un démonstrateur de théorèmes. Sachant qu'il en existe déjà, le plus simple a priori était d'essayer de les utiliser.

Une nouvelle approche consiste maintenant à intégrer l'implémentation d'un démonstrateur de théorèmes dans la version actuelle du système LPG. C'est un peu dans cet esprit que le dernier chapitre de ce document décrit assez brièvement des fonctionnalités de démonstration que l'on souhaiterait avoir à notre disposition.

---

[1]Le transfert du système OASIS sur SUN3 a en fait été réalisé assez récemment.

On peut ainsi imaginer facilement une suite à ce travail dont le thème général serait les algèbres de preuves et la spécification d'un atelier de démonstrations, incorporant cette notion d'algèbre de preuves, en utilisant le langage LPG.

9- Bibliographie

[Bac 78] John Backus
Can Programming Be Liberated from the von Neumann Style ? A Functional Style and Its Algebra of Programs
CACM, vol 21, n°6
1978

[BC 87] Thierry Boy de la Tour, Ricardo Caferra
Proof Analogy in Interactive Theorem Proving : a Method to Express and Use It via Second Order Pattern Matching
A paraître dans les Proceedings de AAAI'87
1987

[BD 87] Didier Bert, Pascal Drabik
LPG : structurations des spécifications et validation sémantique automatisée
BIGRE+GLOBULE, n°55
Juillet 1987

[BDE 87] Didier Bert, Pascal Drabik, Rachid Echahed
Manuel de référence de LPG, version 1.8
RT 17-IMAG 1-LIFIA
Mars 1987

[BDH 86] Leo Bachmair, Nachum Dershowitz, Jiel Hsiang
Orderings for Equational Proofs
Symposium on Logic in Computer Science
Juin 1986

[Ben 85] Saddek Bensalem
Algèbre de Programmes dans un Univers Typé
Thèse de docteur 3° cycle informatique INPG
Décembre 1985

[BE 86] Didier Bert, Rachid Echahed
Design and implementation of a generic, logic and functional programming language
RR 560-IMAG 32-LIFIA
Mars 1986

[BE 87] Didier Bert, Rachid Echahed
Specification and structuration of logic programs in LPG
Rapport interne LIFIA
1987

[Ber 87] Gilles Bernot
Good functors... are those preserving philosophy!
LNCS, n°283
1987

[Bert 83] Didier Bert
Refinements of generic specifications with algebraic tools
Proceedings of the IFIP 9th World Computer Congress, Paris, pp. 815-820
1983

[Bert 87] Didier Bert
Structural operators for inductive programming
Rapport interne LIFIA
1987

[BG 77] R. M. Burstall, J. A. Goguen
Putting theories together to make specifications
Proceedings of 5th International Joint Conference on Artificial Intelligence
Cambridge, pp. 1045-1058
1977

[BJ] G. Barberye, T. Joubert
Un exemple de preuve de représentation à l'aide de l'outil OASIS
NT/PAA/CLC/LSC/1139

[BL 69] R. M. Burstall, P. J. Landin
Programs and their Proofs : an Algebraic Approach
Machine Intelligence, n°4
1969

[BP 68] R. M. Burstall, R. Popplestone
POP-2 reference manual
Machine Intelligence, n°2
1968

[BR 88] Wadoud Bousdira, Jean-Luc Rémy
Hierarchical Contextual Rewriting with several Levels
LNCS, n°294
Février 1988

[Bur 69] R. M. Burstall
Proving properties of programs by structural induction
Computer Journal, vol 12, n°1, pp. 41-48
Février 1969

[Bur 86] R. M. Burstall
Research in Interactive Theorem Proving at Edinburgh University
LFCS Report Series, University of Edinburgh, Department of Computer Science
1986

[Caf 82] Ricardo Caferra
Abstraction, partage de structure et retour arrière non aveugle dans la méthode de réduction matricielle en démonstration automatique de théorèmes
Thèse de docteur 3° cycle informatique USMG
Novembre 1982

[CJ 82] D. Curet, T. Joubert
Spécification formelle d'un noyau temps réel. Validation symbolique d'une réalisation possible. L'exemple de SCEPTRE et GALAXIE
DE/PAA/CLC/LSC/281
Avril 1982

[CW 85] Luca Cardelli, Peter Wegner
On Understanding Types, Data Abstraction and Polymorphism
Computing Surveys, vol 17, n°4
Décembre 1985

[Der 83] N. Dershowitz
Computing with Rewriting Systems
Aerospace Report N° ATR-83 (8478)-1
Janvier 1983

[Dra 88] Pascal Drabik
Exemple de validation sémantique dans les théories structurées
RT 45-IMAG 2-LIFIA
Octobre 1988

[Ech 85] Rachid Echahed
Prédicats et sous-types en LPG : Réalisation de la E-unification
RR 550-IMAG 29-LIFIA
Juillet 1985

[EFH 83] Hartmut Ehrig, Werner Fey, Horst Hansen
ACT ONE : an algebraic specification language with two levels of semantics
Technische Universität Berlin, Technical Report n°83-03
Février 1983

[EM 85] H. Ehrig, B. Mahr
Fundamentals of Algebraic Specification 1; Equations and Initial Semantic
EATCS Monographs on Theoritical Computer Science
1985

[FGMO 87] Kokichi Futatsugi, Joseph Goguen, José Meseguer, Koji Okada
Parameterized Programming in OBJ2
Proceedings of 9th International Conference on Software Engineering, Monterey, USA
1987

[Gan 83] Harald Ganzinger
Parameterized Specifications : Parameter Passing and Implementation with Respect to Obsevability
ACM Transactions on Programming Languages and Systems, vol 5, n°3, pp. 318-354
Juillet 1983

[Gan 86] Harald Ganzinger
A completion procedure for conditional equations : proof of correctness and applications
Report of PROSPECTRA, ESPRIT Project, n°390
1986

[GB 84] J. A. Goguen, R. M. Burstall
Introducing Institutions
Spinger Verlag, Proceedings on Logic of Programming Workshop, pp. 221-256
1984

[Ges 86] Alfons Geser
A specification of the INTEL 8085 microprocessor : a case study
Universität Passau, MIP-8608
Mai 1986

[GH 78] J. V. Guttag, J. J. Horning
The Algebraic Specification of Abstract Data Types
Acta Informatica, n°10, pp.27-52
1978

[GHM 78] John V. Guttag, Ellis Horowitz, David Musser
Abstract Data Types and Software Validation
CACM, vol 21, n°12, pp.1048-1064
Décembre 1978

[GM 81] J. A. Goguen, J. Meseguer
Completness of many-sorted equational logic
SIGPLAN Notice, vol 16, n°7, pp. 24-32
1981

[GM 83] Joseph A. Goguen and José Meseguer
Initiality, Induction, and Computability
Application of Algebra to Language Definition and Compilation, M. Nivat and J. Reynolds (eds.), Cambridge U.P.
1984

[GM 87a] Joseph A. Goguen and José Meseguer
Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems
Proceedings on Logic in Computer Science
Juin 1987

[GM 87b] Joseph A. Goguen, José Meseguer
Remarks on Remarks on Many-Sorted Equationnal Logic
SIGPLAN Notices, vol 22, n°4
Avril 1987

[GMW 79] Michael J. Gordon, Arthur J. Milner, Christopher P. Wadsworth
Edinburgh LCF
LNCS, n°78
1979

[Gor 81] Michael J. Gordon
Representing a Logic in the LCF Metalanguage
Tools & Notions for Program Construction, D. Neel, Cambridge
Décembre 1981

[Gog 80] Joseph A. Goguen
How to prove algebraic inductive hypotheses without induction with applications to the correctness of data type implementation
Proceedings of 5th CADE
1980

[Gog 88] Joseph A. Goguen
OBJ as a Theorem Prover with Applications to Hardware Verification
SRI-CSL-88-4R2
August 1988

[GTW 78] J. A. Goguen, J. W. Thatcher, E. G. Wagner
An initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types
Current Trends in Programming Methodology, vol 4, Data Structuring, Chap. 5, Prentice Hall
1978

[GTWW 77] J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright
Initial algebra semantics and continuous algebras
JACM, n°24, pp. 68-95
1977

[Gut 77] John Guttag
Abstract Data Types and the Development of Data Structures
CACM, vol 20, n°6
Juin 1977

[Hue 80] Gérard Huet
A complete proof of correctness of the Knuth-Bendix completion algorithm
Rapport de recherche INRIA, n°25
Juillet 1980

[Hue 86] Gérard Huet
Formal Structures for Computation and Deduction
Mai 1986

[Hul 80] Jean-Marie Hullot
Canonical forms and unification
Proceedings of 5th CADE, LNCS, n°87, pp. 318-334
1980

[JK 86] J. P. Jouannaud, E. Kounalis
Automatic Proofs by Induction in Theories without Constructors
Rapport de Recherche LRI, n°295
Septembre 1986

[Jou 82] Thierry Joubert
Exemples de spécifications abstraites de types et de représentations de types en OASIS
DE/CLC/LSC/33
1982

[Kla 84] Herbert A. Klaeren
A Constructive Method for Abstract Algebraic Software Specification
Theoritical Computer Science, n°30
Août 1984

[Klo 87] J. W. Klop
Term Rewrinting Systems
Tutorial for ESPRIT project 415 WGS
Février 1987

[Kow 79] Robert Kowalski
Algorithm = Logic + Control
CACM, vol 22, n°7
Juillet 1979

[LB 88] B. Lampson, R. Burstall
Pebble, a Kernel Language for Modules and Abstract Data Types
Information and Computation, n°76, pp. 278-346
1988

[LZ 74] Barbara Liskov, Stephen Zilles
Programming with Abstract Data Types
Proceedings of a Symposium on very high level languages,
SIGPLAN Notices, vol 9, n°4
Avril 1974

[MMN 75] R. Milner, L. Morris, M. Newey
A logic for Computable Functions with reflexive and polymorphic types
Proceedings Conference on Proving and Improving Programs
1975

[MNV 72] Zohar Manna, Stephen Ness, Jean Vuillemin
Inductive methods for proving properties of programs
Proceeding of an ACM conference on Proving Assertions about Programs
SIGPLAN Notices, vol 7, n°1
Janvier 1972

[Mus 80a] David R. Musser
Abstract Data Type Specification in the AFFIRM System
IEEE Transactions on Software Engineering, vol SE-6, n°1
Janvier 1980

[Mus 80b] David R. Musser
On proving Inductive Properties of Abstract Data Types
Proceedings of the 7th ACM Symposium on Principle Languages
Janvier 1980

[NS 87] Jean-Paul Nicollin, Xavier Schmidt
Exemples de spécifications en LPG
Rapport interne LIFIA
Septembre 1987

[Pad 84] Peter Padawitz
Towards a proof theory of parameterized specifications
LNCS, n°173, pp. 375-391
1984

[Pad 85] Peter Padawitz
Parameter Preserving Data Type Specification
LNCS, n°185, pp. 323-341
1985

[Pad 88] Peter Padawitz
The Equational Theory of Parameterized Specifications
Information and Computation, n°76, pp. 121-137
1988

[Pau 83a] Lawrence Paulson
Recent developments in LCF : examples of structural induction
Report, Computer Laboratory, University of Cambridge, n°34
1983

[Pau 83b] Lawrence Paulson
The revised logic PPLAMBDA : a reference manual
Report, Computer Laboratory, University of Cambridge, n°36
1983

[Pau 83c] Lawrence Paulson
Tactics and tacticals in Cambridge LCF
Report, Computer Laboratory, University of Cambridge, n°39
1983

[Pau 84a] Lawrence Paulson
Structural Induction in LCF
Report, Computer Laboratory, University of Cambridge, n°44
Février 1984

[Pau 84b] Lawrence Paulson
Lessons learned from LCF
Report, Computer Laboratory, University of Cambridge, n°54
1984

[Pau 88] Lawrence Paulson
The Foundation of a Generic Theorem Prover
Technical Report, University of Cambridge, Computer Laboratory, n°130
Mars 1988

[Paul 83] Etienne Paul
Spécification formelle de la fonction "Traduction téléphonique" (Partie "analyse")
NT/PAA/CLC/LSC/939
1983

[Paul 84] Etienne Paul
Proof by Induction in Equational Theories with Relations between Constructors
Proceedings of CAAP'84
Mars 1984

[Paul 85] Etienne Paul
Manuel de OASIS (version février 1985)
NT/PAA/CLC/LSC/959
Février 1985

[Ren 85] Eric Renard
Perfectionnement du démonstrateur de théorème de l'outil OASIS d'aide à la spécification
DE/CLC/LSC/623
1985

[Rey 87] Jean-Claude Reynaud
Sémantique de LPG
RR 651-IMAG 56-LIFIA
Mars 1987

[Reynolds 70] J. C. Reynolds
GEDANKEN : a simple typeless programming language based on the principle of completness and the reference concept
CACM, vol 13, n°5
Mai 1970

[RW 69] G. Robinson, L. Wos
Paramodulation and Theorem-proving in First-Order Theories with Equality
Machine Intelligence, n°4, 1969
1969

[SB 83] D. T. Sannella, R. M. Burstall
Structured Theories in LCF
Internal Report CSR-129-83, University of Edinburgh, Department of Computer Science
Février 1983

[Sok 83] Stefan Sokolowski
A Note on Tactics in LCF
Internal Report CSR-140-83, University of Edinburgh, Department of Computer Science
1983

[STEGS 82] Carl A. Sunshine, David H. Thompson, Roddy W. Erickson, Susan L. Gerhart, Daniel Schwabe
Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models
IEEE Transactions on Software Engineering, vol SE-8, n°5
Septembre 1982

[SWGC 88] M. Sintzoff, M. Weber, Ph. de Groote, J. Cazin
Definition 0.1 of the approximation DEVA.0 of a development language
ToolUse ESPRIT Project ToolUse.TD.deva01.DD88a
1988

[TWW 82] J. W. Thatcher, E. J. Wagner, J. B. Wright
Data Type Specification : Parameterization and the Power of Specification Techniques
ACM Transactions on Programming Languages and Systems, vol 4, 4 pp.711-732
Octobre 1982

[Van 88] Joëlle Vangeersdael
A Guided Tour through Theorem Provers
Report of ATES, ESPRIT Project, n°1158
Février 1988

10- Annexes

10.1- Contenu des fichiers OASIS

Nous présentons dans cette partie le contenu des différents fichiers manipulés par OASIS. Une solution d'interfaçage des logiciels aurait pu consister à utiliser ces fichiers comme intermédiaire entre les deux systèmes.

On peut dire que pour un objet OASIS, quatre fichiers peuvent lui être associés. Cette annexe est destinée à montrer la structure interne des fichiers de type *SO*, *SSO* et *SPO*. L'exemple choisi est celui de la spécification algébrique des listes. Pour chaque type de fichier (le fichier contenant la spécification externe mis à part), nous montrerons le contenu même du fichier, qui est codé à l'usage de l'interpréteur PROLOG et une image sous forme de clauses de Horn de ce contenu.

10.1.1- Fichier de type SE

C'est dans ce type de fichier que l'on retrouve le texte original de toute spécification de type abstrait en OASIS.

```
type LISTE =
import
    BOOL.
constantes
    NULL : LISTE.
    .
operations
    CONS(LISTE,LISTE)  ->  LISTE.
    APPEND(LISTE,LISTE)  ->  LISTE.
    REV(LISTE)  ->  LISTE.
    .
var_eqs
    x,y,z : LISTE.
    .
equations
    1 : APPEND(NULL,x) == x.
    2 : APPEND(CONS(x,y),z) == CONS(x,APPEND(y,z)).
    3 : REV(NULL) == NULL.
    4 : REV(CONS(x,y)) == APPEND(REV(y),CONS(x,NULL)).
    .
var_theo
    x,y : LISTE.
    .
schema_induc
```

```
    1 : P(x) <= ( P(NULL) et ((P(x) et P(y)) =>
                                P(CONS(x,y)))).
    .
theoremes
    1 : REV(REV(x)) = x.
    .
fin type LISTE.
.
```

10.1.2- Fichier de type SO

Ce type de fichier contient la forme interne de la spécification.

```
D1+5+LISTEI1+0+P0+I1+0+D2+4+TYPEI2+0+AC0+I1+0+D3+6+IMPORTI3+0+AD4+4+BOOLI4+0+AC
0+I1+0+D5+3+CSTI5+0+AD6+4+NULLI6+0+AI1+0+AC0+I1+0+D7+3+OPSI7+0+AD8+4+CONSI8+0+I
1+0+AI1+0+AAI1+0+AC0+I1+0+I7+0+AD9+6+APPENDI9+0+I1+0+AI1+0+AAI1+0+AC0+I1+0+I7+0
+AD:+3+REVI:+0+I1+0+AAI1+0+AC0+I1+0+D;+6+VAREQSI;+0+AD<+3+'x'I<+0+AI1+0+AC0+I1+
0+I;+0+AD=+3+'y'I=+0+AI1+0+AC0+I1+0+I;+0+AD>+3+'z'I>+0+AI1+0+AC0+I1+0+D?+3+EQSI
?+0+E1+SAI9+0+I6+0+AI<+0+AAI<+0+AC0+I1+0+I?+0+E2+SAI9+0+I8+0+I<+0+AI=+0+AAI>+0+
AAI8+0+I<+0+AI9+0+I=+0+AI>+0+AAAC0+I1+0+I?+0+E3+SAI:+0+I6+0+AAI6+0+AC0+I1+0+I?+
0+E4+SAI:+0+I8+0+I<+0+AI=+0+AAAI9+0+I:+0+I=+0+AAI8+0+I<+0+AI6+0+AAAC0+I1+0+D@+7
+VARTHEOI@+0+AI<+0+AI1+0+AC0+I1+0+I@+0+AI=+0+AI1+0+AC0+I1+0+DA+6+SCHEMAIA+0+E1+
SADB+4+'<='IB+0+DC+3+'$'IC+0+SDD+1+PID+0+I<+0+AADE+4+'et'IE+0+IC+0+SID+0+I6+0+A
ADF+4+'=>'IF+0+IC+0+SIE+0+IC+0+SID+0+I<+0+AAID+0+I=+0+AAAID+0+I8+0+I<+0+AI=+0+A
AAAAAC0+I1+0+DG+4+THEOIG+0+E1+SADH+3+'='IH+0+IC+0+SI:+0+I:+0+I<+0+AAAI<+0+AAC0+
Q
```

Voici l'équivalent, décodé, sous forme de clauses de Horn :

```
LISTE(TYPE).
LISTE(IMPORT,BOOL).
LISTE(CST,NULL,LISTE).
LISTE(OPS,CONS(LISTE,LISTE),LISTE).
LISTE(OPS,APPEND(LISTE,LISTE),LISTE).
LISTE(OPS,REV(LISTE),LISTE).
LISTE(VAREQS,'x',LISTE).
LISTE(VAREQS,'y',LISTE).
LISTE(VAREQS,'z',LISTE).
LISTE(EQS.1,APPEND(NULL,'x'),'x').
LISTE(EQS.2,APPEND(CONS('x','y'),'z'),CONS('x',APPEND('y','z'))).
LISTE(EQS.3,REV(NULL),NULL).
LISTE(EQS.4,REV(CONS('x','y')),APPEND(REV('y'),CONS('x',NULL))).
LISTE(VARTHEO,'x',LISTE).
LISTE(VARTHEO,'y',LISTE).
LISTE(SCHEMA.1,('<='.'$')(P('x'),('et'.'$')(P(NULL),('=>'.'$')(('et'.'$')(P('x'),P('y')
),P(CONS('x','y')))))).
LISTE(THEO.1,('='.'$')(REV(REV('x')),'x')).
```

10.1.3- Fichier de type SSO

Ce type de fichier contient les règles de syntaxe (profiles) des opérateurs de la spécification ainsi que la liste des opérateurs définis.

```
D1+1+SI1+0+P2-I1+0+D2+4+NULLI2+0+AD3+5+LISTEI3+0+AC0+I1+0+D4+4+CONSI4+0+I3+0+AI
3+0+AAI3+0+AC0+I1+0+D5+6+APPENDI5+0+I3+0+AI3+0+AAI3+0+AC0+I1+0+D6+3+REVI6+0+I3+
0+AAI3+0+AC0+I1+0+D7+4+VRAII7+0+AD8+4+BOOLI8+0+AC0+I1+0+D9+4+FAUXI9+0+AI8+0+AC0
+I1+0+D:+5+'non'I:+0+I8+0+AAI8+0+AC0+I1+0+D;+4+'ou'I;+0+D<+3+'$'I<+0+SI8+0+AI8+
0+AAI8+0+AC0+I1+0+D=+4+'et'I=+0+I<+0+SI8+0+AI8+0+AAI8+0+AC0+I1+0+D>+4+'=>'I>+0+
I<+0+SI8+0+AI8+0+AAI8+0+AC0+I1+0+D?+5+'<=>'I?+0+I<+0+SI8+0+AI8+0+AAI8+0+AC0+I1+
0+D@+4+'<>'I@+0+I<+0+SDA+2+*1VA+1+AVA+1+AAI8+0+AC1+I1+0+DB+3+'='IB+0+I<+0+SVA+1
+AVA+1+AAI8+0+AC1+I1+0+DC+2+SIIC+0+I8+0+AVA+1+AVA+1+AAVA+1+AC1+I1+0+DD+4+CONDID
+0+I8+0+AVA+1+AAVA+1+AC1+Q

DE+:+OPERATEURSIE+0+P2-IE+0+DF+6+'NULL'IF+0+AC0+IE+0+DG+6+'CONS'IG+0+AC0+IE+0+D
H+8+'APPEND'IH+0+AC0+IE+0+DI+5+'REV'II+0+AC0+IE+0+DJ+6+'VRAI'IJ+0+AC0+IE+0+DK+6
+'FAUX'IK+0+AC0+IE+0+I:+0+AC0+IE+0+I;+0+AC0+IE+0+I=+0+AC0+IE+0+I>+0+AC0+IE+0+I?
+0+AC0+IE+0+I@+0+AC0+IE+0+IB+0+AC0+IE+0+DL+4+'SI'IL+0+AC0+IE+0+DM+6+'COND'IM+0+
AC0+IE+0+DN+3+'P'IN+0+AC0+Q
```

Sous une forme décodée, plus lisible :

```
S(NULL,LISTE).
S(CONS(LISTE,LISTE),LISTE).
S(APPEND(LISTE,LISTE),LISTE).
S(REV(LISTE),LISTE).
S(VRAI,BOOL).
S(FAUX,BOOL).
S('non'(BOOL),BOOL).
S(('ou'.'$')(BOOL,BOOL),BOOL).
S(('et'.'$')(BOOL,BOOL),BOOL).
S(('=>'.'$')(BOOL,BOOL),BOOL).
S(('<=>'.'$')(BOOL,BOOL),BOOL).
S(('<>'.'$')(*1,*1),BOOL).
S(('='.'$')(*1,*1),BOOL).
S(SI(BOOL,*1,*1),*1).
S(COND(BOOL,*1),*1).

OPERATEURS('NULL').
OPERATEURS('CONS').
OPERATEURS('APPEND').
OPERATEURS('REV').
OPERATEURS('VRAI').
OPERATEURS('FAUX').
OPERATEURS('non').
```

```
OPERATEURS('ou').
OPERATEURS('et').
OPERATEURS('=>').
OPERATEURS('<=>').
OPERATEURS('<>').
OPERATEURS('=').
OPERATEURS('SI').
OPERATEURS('COND').
OPERATEURS('P').
```

10.1.4- Fichier de type SPO

Ce type de fichier contient les différentes règles de réécriture définies.

```
D1+6+'NULL'I1+0+P2-Q

D2+6+'CONS'I2+0+P2-Q

D3+8+'APPEND'I3+0+P2-I3+0+D4+6+APPENDI4+0+D5+4+NULLI5+0+AD6+2+*1V6+1+AAV6+1+AC1
+I3+0+I4+0+D7+4+CONSI7+0+V6+1+AD8+2+*2V8+2+AAD9+2+*3V9+3+AAI7+0+V6+1+AI4+0+V8+2
+AV9+3+AAAC3+Q

D:+5+'REV'I:+0+P2-I:+0+D;+3+REVI;+0+I5+0+AAI5+0+AC0+I:+0+I;+0+I7+0+V6+1+AV8+2+A
AAI4+0+I;+0+V8+2+AAI7+0+V6+1+AI5+0+AAAC2+Q

D<+6+'VRAI'I<+0+P2-Q

D=+6+'FAUX'I=+0+P2-Q

D>+5+'non'I>+0+P2-I>+0+I>+0+D?+4+VRAII?+0+AAD@+4+FAUXI@+0+AC0+I>+0+I>+0+I@+0+AA
I?+0+AC0+I>+0+I>+0+I>+0+V6+1+AAAV6+1+AC1+Q

DA+4+'ou'IA+0+P2-IA+0+IA+0+DB+3+'$'IB+0+SI?+0+AV9+1+AAI?+0+AC1+IA+0+IA+0+IB+0+S
V8+1+AI?+0+AAI?+0+AC1+IA+0+IA+0+IB+0+SI@+0+AV9+1+AAV9+1+AC1+IA+0+IA+0+IB+0+SV8+
1+AI@+0+AAV8+1+AC1+IA+0+IA+0+IB+0+SV8+1+AV8+1+AAV8+1+AC1+IA+0+IA+0+IB+0+SIA+0+I
B+0+SV8+1+AV9+2+AAV9+2+AAIA+0+IB+0+SV8+1+AV9+2+AAC2+IA+0+IA+0+IB+0+SV6+1+AI>+0+
V6+1+AAAI?+0+AC1+IA+0+IA+0+IB+0+SI>+0+V6+1+AAV6+1+AAI?+0+AC1+Q

DC+4+'et'IC+0+P2-IC+0+IC+0+IB+0+SI?+0+AV9+1+AAV9+1+AC1+IC+0+IC+0+IB+0+SV8+1+AI?
+0+AAV8+1+AC1+IC+0+IC+0+IB+0+SI@+0+AV9+1+AAI@+0+AC1+IC+0+IC+0+IB+0+SV8+1+AI@+0+
AAI@+0+AC1+IC+0+IC+0+IB+0+SV8+1+AV8+1+AAV8+1+AC1+IC+0+IC+0+IB+0+SIC+0+IB+0+SV8+
1+AV9+2+AAV9+2+AAIC+0+IB+0+SV8+1+AV9+2+AAC2+IC+0+IC+0+IB+0+SV6+1+AI>+0+V6+1+AAA
I@+0+AC1+IC+0+IC+0+IB+0+SI>+0+V6+1+AAV6+1+AAI@+0+AC1+Q

DD+4+'=>'ID+0+P2-ID+0+ID+0+IB+0+SI?+0+AV9+1+AAV9+1+AC1+ID+0+ID+0+IB+0+SI@+0+AV9
```

```
+1+AAI?+0+AC1+ID+0+ID+0+IB+0+SV8+1+AI?+0+AAI?+0+AC1+ID+0+ID+0+IB+0+SV8+1+AI@+0+
AAI>+0+V8+1+AAC1+ID+0+ID+0+IB+0+SV8+1+AV8+1+AAI?+0+AC1+ID+0+ID+0+IB+0+SV8+1+AIC
+0+IB+0+SV8+1+AV9+2+AAAID+0+IB+0+SV8+1+AV9+2+AAC2+ID+0+ID+0+IB+0+SV8+1+AIC+0+IB
+0+SV9+2+AV8+1+AAAID+0+IB+0+SV8+1+AV9+2+AAC2+Q

DE+5+'<=>'IE+0+P2-IE+0+IE+0+IB+0+SV8+1+AV8+1+AAI?+0+AC1+IE+0+IE+0+IB+0+SI?+0+AV
9+1+AAV9+1+AC1+IE+0+IE+0+IB+0+SV8+1+AI?+0+AAV8+1+AC1+IE+0+IE+0+IB+0+SI@+0+AV9+1
+AAI>+0+V9+1+AAC1+IE+0+IE+0+IB+0+SV8+1+AI@+0+AAI>+0+V8+1+AAC1+Q

DF+4+'<>'IF+0+P2-IF+0+IF+0+IB+0+SDG+2+*5VG+1+ADH+2+*6VH+2+AAI>+0+DI+3+'='II+0+I
B+0+SVG+1+AVH+2+AAAC2+Q

II+0+P2-II+0+II+0+IB+0+SI?+0+AI@+0+AAI@+0+AC0+II+0+II+0+IB+0+SI@+0+AI?+0+AAI@+0
+AC0+Q

DJ+4+'SI'IJ+0+P2-Q

DK+6+'COND'IK+0+P2-Q

DL+3+'P'IL+0+P2-Q

DM+2+PRIM+0+P2-Q

DN+:+regles_symIN+0+P2-Q
```

Soit après décodage sous forme de clauses de Horn :

```
'APPEND'(APPEND(NULL,*1),*1).
'APPEND'(APPEND(CONS(*1,*2),*3),CONS(*1,APPEND(*2,*3))).
'REV'(REV(NULL),NULL).
'REV'(REV(CONS(*1,*2)),APPEND(REV(*2),CONS(*1,NULL))).
'non'('non'(VRAI),FAUX).
'non'('non'(FAUX),VRAI).
'non'('non'('non'(*1)),*1).
'ou'(('ou'.'$')(VRAI,*3),VRAI).
'ou'(('ou'.'$')(*2,VRAI),VRAI).
'ou'(('ou'.'$')(FAUX,*3),*3).
'ou'(('ou'.'$')(*2,FAUX),*2).
'ou'(('ou'.'$')(*2,*2),*2).
'ou'(('ou'.'$')(('ou'.'$')(*2,*3),*3),('ou'.'$')(*2,*3)).
'ou'(('ou'.'$')(*1,'non'(*1)),VRAI).
'ou'(('ou'.'$')('non'(*1),*1),VRAI).
'et'(('et'.'$')(VRAI,*3),*3).
'et'(('et'.'$')(*2,VRAI),*2).
'et'(('et'.'$')(FAUX,*3),FAUX).
'et'(('et'.'$')(*2,FAUX),FAUX).
```

```
'et'(('et'.'$')(*2,*2),*2).
'et'(('et'.'$')(('et'.'$')(*2,*3),*3),('et'.'$')(*2,*3)).
'et'(('et'.'$')(*1,'non'(*1)),FAUX).
'et'(('et'.'$')('non'(*1),*1),FAUX).
'=>'(('=>'.'$')(VRAI,*3),*3).
'=>'(('=>'.'$')(FAUX,*3),VRAI).
'=>'(('=>'.'$')(*2,VRAI),VRAI).
'=>'(('=>'.'$')(*2,FAUX),'non'(*2)).
'=>'(('=>'.'$')(*2,*2),VRAI).
'=>'(('=>'.'$')(*2,('et'.'$')(*2,*3)),('=>'.'$')(*2,*3)).
'=>'(('=>'.'$')(*2,('et'.'$')(*3,*2)),('=>'.'$')(*2,*3)).
'<=>'(('<=>'.'$')(*2,*2),VRAI).
'<=>'(('<=>'.'$')(VRAI,*3),*3).
'<=>'(('<=>'.'$')(*2,VRAI),*2).
'<=>'(('<=>'.'$')(FAUX,*3),'non'(*3)).
'<=>'(('<=>'.'$')(*2,FAUX),'non'(*2)).
'<>'(('<>'.'$')(*5,*6),'non'(('='.'$')(*5,*6))).
'='(('='.'$')(VRAI,FAUX),FAUX).
'='(('='.'$')(FAUX,VRAI),FAUX).
.
```

10.2- Bibliothèque du système OASIS

Nous présentons dans cette partie la signature des types abstraits de la bibliothèque du système OASIS. Pour une version complète, on peut consulter [Paul 85].

D'une manière générale, les sortes *ELEM*, *ELEM1*, *ELEM2*, *NOEUD* et *INDICE* seront des sortes paramètres de la signature qui les utilise.

Le nom des constantes sera suivi de *: <name_sort>* qui indique de quelle sorte est la constante. La sorte du résultat d'un opérateur sera notée -> *<name_sort>*.

10.2.1- bool

Il n'est pas possible de redéfinir le nom de ce type, ni le nom des deux constantes, ni même le nom de ces opérations du fait de leur utilisation par le système de réécriture.

VRAI,FAUX : *BOOL*
non(*BOOL*) -> *BOOL*
BOOL ou *BOOL* -> *BOOL*
BOOL et *BOOL* -> *BOOL*
BOOL => *BOOL* -> *BOOL*
BOOL <=> *BOOL* -> *BOOL*
ELEM <> *ELEM* -> *BOOL*
ELEM = *ELEM* -> *BOOL*
si *BOOL* alors *ELEM* sinon *ELEM* -> *ELEM*
si *BOOL* alors *ELEM* -> *ELEM*

10.2.2- entier

De même que pour le type des booléens, on ne peut redéfinir ni le nom de ce type, ni celui de ses opérations et constantes.

Il a été prévu d'utiliser les constantes entières traditionnelles, ce qui fait que l'opérateur *succ* constructeur des entiers n'apparait pas dans cette signature.

```
0,1 : ENTIER
ENTIER + ENTIER -> ENTIER
ENTIER - ENTIER -> ENTIER
-(ENTIER) -> ENTIER
ENTIER * ENTIER -> ENTIER
ENTIER / ENTIER -> ENTIER
mod(ENTIER,ENTIER) -> ENTIER
ENTIER > ENTIER -> BOOL
ENTIER < ENTIER -> BOOL
ENTIER >= ENTIER -> BOOL
ENTIER =< ENTIER -> BOOL
```

10.2.3- enrich_entier

Théoriquement, cette signature correspond à la définition d'un nouveau type abstrait qui défini la nouvelle sorte *ENRICH_ENTIER* (sans d'ailleurs qu'il y ait d'opérateur dont le résultat est cette sorte) ainsi que deux opérateurs sur les entiers. Pratiquement, elle ne fait qu'introduire deux nouvelles fonctions sur les valeurs entières.

```
fact(ENTIER) -> ENTIER                  * Factorielle
ENTIER ** ENTIER -> ENTIER              * Elevation a la puissance
```

10.2.4- couple

Spécification du produit cartésien de deux sortes.

```
ELEM1 x ELEM2 -> COUPLE(ELEM1,ELEM2)
gauche(COUPLE(ELEM1,ELEM2)) -> ELEM1
droite(COUPLE(ELEM1,ELEM2)) -> ELEM2
```

10.2.5- seq

Spécification des séquences.

```
SEQ_VIDE : SEQ(ELEM)
SEQ(ELEM) ; ELEM -> SEQ(ELEM)           * Ajout a droite
ELEM ;; SEQ(ELEM) -> SEQ(ELEM)          * Ajout a gauche
s(ELEM) -> SEQ(ELEM)                    * Fonction list de LISP
SEQ(ELEM) & SEQ(ELEM) -> SEQ(ELEM)      * Concatenation
debut(SEQ(ELEM)) -> SEQ(ELEM)           * Reste dans le cas ajout
                                        *   a droite
fin(SEQ(ELEM)) -> SEQ(ELEM)             * Suppression du dernier
                                        *   element
iemes_premiers(SEQ(ELEM),ENTIER) -> SEQ(ELEM)
```

```
iemes_derniers(SEQ(ELEM),ENTIER) -> SEQ(ELEM)
dedup(SEQ(ELEM)) -> SEQ(ELEM)              * Suppression des elements
                                           *   repetes
inverse(SEQ(ELEM)) -> SEQ(ELEM)
circule(SEQ(ELEM),ENTIER) -> SEQ(ELEM)
ieme(SEQ(ELEM),ENTIER) -> ELEM             * ieme element de la
                                           *   sequence
retire(SEQ(ELEM),ENTIER) -> SEQ(ELEM)
vide(SEQ(ELEM)) -> BOOL
SEQ(ELEM) [ SEQ(ELEM) -> BOOL              * Inclusion
ELEM dans SEQ(ELEM) -> BOOL
nodup(SEQ(ELEM)) -> BOOL
SEQ(ELEM) disjointes SEQ(ELEM) -> BOOL
taille(SEQ(ELEM)) -> ENTIER
premier(SEQ(ELEM)) -> ELEM
dernier(SEQ(ELEM)) -> ELEM
P(ELEM) -> BOOL                            * Declaration du profile
                                           *   d'un predicat qui sera
                                           *   specifie par ailleurs
TOUSP(SEQ(ELEM)) -> BOOL                   * Tous les elements de la
                                           *   sequence verifient le
                                           *   predicat
DENONP(SEQ(ELEM)) -> SEQ(ELEM)             * Suppression des elements
                                           *   qui ne vérifient pas
                                           *   le predicat
```

10.2.6- ensemble

Spécification du type des ensembles.

```
ENS_VIDE : ENSEMBLE(ELEMEM)
ENSEMBLE(ELEM) u ELEM -> ENSEMBLE(ELEM) * Ajout d'un element
enlever(ENSEMBLE(ELEM),ELEM) -> ENSEMBLE(ELEM)
vide(ENSEMBLE(ELEM)) -> BOOL
ELEM c ENSEMBLE(ELEM) -> BOOL              * Appartenance
ENSEMBLE(ELEM) C ENSEMBLE(ELEM) -> BOOL * Inclusion
taille(ENSEMBLE(ELEM)) -> ENTIER
ENSEMBLE(ELEM) U ENSEMBLE(ELEM) -> ENSEMBLE(ELEM)
ENSEMBLE(ELEM) ^ ENSEMBLE(ELEM) -> ENSEMBLE(ELEM)
```

10.2.7- pile

Spécification des piles.

```
PILE_VIDE : PILE(ELEM)
empiler(ELEM,PILE(ELEM)) -> PILE(ELEM)
depiler(PILE(ELEM)) -> PILE(ELEM)
sommet(PILE(ELEM)) -> ELEM
vide(PILE(ELEM)) -> BOOL
taille(PILE(ELEM)) -> ENTIER
```

remplacer(*ELEM*, *PILE*(*ELEM*)) -> *PILE*(*ELEM*)

10.2.8- vecteur

Spécification des vecteurs.

VECTEUR_VIDE : *VECTEUR*(*ELEM*, *INDICE*)
modifier(*VECTEUR*(*ELEM*, *INDICE*), *INDICE*, *ELEM*) -> *VECTEUR*(*ELEM*, *INDICE*)
valeur(*VECTEUR*(*ELEM*, *INDICE*), *INDICE*) -> *ELEM*

10.2.9- file

Spécification de la structure de file (*FIFO*).

FILE_VIDE : *FILE*(*ELEM*)
ajouter(*ELEM*, *FILE*(*ELEM*)) -> *FILE*(*ELEM*)
retirer(*FILE*(*ELEM*)) -> *FILE*(*ELEM*)
premier(*FILE*(*ELEM*)) -> *ELEM*
vide?(*FILE*(*ELEM*)) -> *BOOL*
taille(*FILE*(*ELEM*)) -> *ENTIER*

10.2.10- liste

Spécification des listes.

NULL : *LISTE*
CONS(*LISTE*, *LISTE*) -> *LISTE*
APPEND(*LISTE*, *LISTE*) -> *LISTE*
REV(*LISTE*) -> *LISTE*

10.2.11- groupe

Caractérisation d'une structure de groupe.

NEUTRE : *GROUPE*
i(*GROUPE*) -> *GROUPE*
GROUPE x *GROUPE* -> *GROUPE*

10.2.12- graphe

Spécification du type abstrait des graphes.

GRAPHE_VIDE : *GRAPHE*(*NOEUD*)
aj_arc(*GRAPHENOEUD*), *NOEUD*, *NOEUD*) -> *GRAPHENOEUD*)
aj_arcs_vers_noeud(*GRAPHENOEUD*), *ENSEMBLE*(*NOEUD*), *NOEUD*)
-> *GRAPHENOEUD*)
aj_arcs_depuis_noeud(*GRAPHENOEUD*), *ENSEMBLE*(*NOEUD*), *NOEUD*)
-> *GRAPHENOEUD*)
NOEUD est_dans *GRAPHENOEUD*) -> *BOOL*
GRAPHENOEUD) a_un_arc_de *NOEUD* a *NOEUD* -> *BOOL*
NOEUD est_une_feuille_dans *GRAPHENOEUD*) -> *BOOL*
NOEUD est_sans_fils_dans *GRAPHENOEUD*) -> *BOOL*
GRAPHENOEUD) a_un_chemin_de *NOEUD* a *NOEUD* -> *BOOL*
SEQ(*NOEUD*) est_un_chemin_de *GRAPHENOEUD*) -> *BOOL*

```
SEQ(NOEUD) est_un_chemin_de NOEUD a NOEUD dans GRAPHENOEUD)
                        -> BOOL
NOEUD est_un_noeud_du_chemin SEQ(NOEUD) de GRAPHENOEUD) -> BOOL
GRAPHENOEUD) est_une_extension_de GRAPHENOEUD) -> BOOL
GRAPHENOEUD) est_une_extension_disjointe_de GRAPHENOEUD)
                        -> BOOL
vide(GRAPHENOEUD)) -> BOOL
noeuds(GRAPHENOEUD)) -> ENSEMBLE(NOEUD)
feuilles(GRAPHE(NOEUD)) -> ENSEMBLE(NOEUD)
fils_de(GRAPHE(NOEUD),NOEUD) -> ENSEMBLE(NOEUD)
parents_de(GRAPHE(NOEUD),NOEUD) -> ENSEMBLE(NOEUD)
nbre_noeuds(GRAPHE(NOEUD)) -> ENTIER
nbre_arcs(GRAPHE(NOEUD)) -> ENTIER
```

10.3- Bibliothèque du système LPG

Cette bibliothèque prédéfinie est composée de deux parties (deux fichiers). Dans la première partie, on trouve des déclarations un peu particulières dans le sens où elles ne sont pas absolument correctes d'un point de vue spécification (définition de type sans constructeurs, etc...). Ces incohérences sont voulues et dues au fait que certains opérateurs sont directement implantés (cablés) dans le système LPG pour des rasons d'efficacité. Le deuxième fichier, quant à lui, contient des définitions normales venant enrichir initialement le système.

Nous allons donner les signatures des différents modules figurant dans cette bibliothèque prédéfinie et par catégorie de module. Pour chaque module, on donne tout d'abord son nom puis les sortes qu'il introduit. On liste ensuite les opérateurs avec, pour chacun d'eux son nom et son profile. Les constantes sont caractérisées par le fait qu'elles n'ont pas de domaine.

10.3.1- Les propriétés

Caractérisation d'une sorte formelle.

```
Formal_Sort   t
```

Caractérisation de la commutativité d'un opérateur.

```
Com  t
     + : (t,t) -> t
```

Associativité d'un opérateur.

```
Assoc  t
       + : (t,t) -> t
```

Structure de monoïde. Il est nécessaire de définir un opérateur et une constante.

```
Monoid t
       z :        -> t
       + : (t,t)  -> t
```

Caractérisation de l'associativité et de la commutativité d'un opérateur.

```
AC   t
     + : (t,t) -> t
```

Structure de monoïde commutatif.

```
MC   t
     z :        -> t
     + : (t,t)  -> t
```

Spécification d'un opérateur d'égalité.

```
Equality t
         = : (t,t) -> bool
```

Spécification d'une relation d'ordre partielle. La présence de l'opérateur d'égalité est justifiée par son utilisation dans la définition de la relation.

```
Partial_Order t
              <=   : (t,t) -> bool
              =    : (t,t) -> bool
```

Description d'une relation d'ordre totale. Même remarque que précédemment en ce qui concerne la relation d'égalité.

```
Total_Order   t
              <=   : (t,t) -> bool
              =    : (t,t) -> bool
```

10.3.2- Les types

Spécification du type des booléens. On peut noter l'absence des constantes *vrai* et *faux* qui sont en fait implémantées (cablées) dans le langage.

```
Bool   bool
       not  : bool        -> bool
       and  : (bool,bool) -> bool
       or   : (bool,bool) -> bool
       xor  : (bool,bool) -> bool
       =>   : (bool,bool) -> bool
```

Spécification des entiers naturels. Même remarque quant à la constante *0*.

```
Natural  nat
         succ : nat        -> nat
         pred : nat        -> nat
         +    : (nat,nat)  -> nat
         -    : (nat,nat)  -> nat
         *    : (nat,nat)  -> nat
```

Spécification du type des caractères. Les constantes ne figurent pas car elles sont également implantées dans le langage.

```
Character  char
           succ    : char        -> char
           pred    : char        -> char
           ord     : char        -> nat
           ordinv  : nat         -> char
           =       : (char,char) -> bool
           /=      : (char,char) -> bool
           <=      : (char,char) -> bool
           <       : (char,char) -> bool
```

```
>=      : (char,char) -> bool
>       : (char,char) -> bool
```

Description du type des chaînes de caractères.

```
String string
        <+          : (char,string)        -> string
        +>          : (string,char)        -> string
        cv_string   : char                 -> string
        cv_char     : string               -> char
        #           : string               -> nat
        +           : (string,string)      -> string
        ||          : (string,(nat,nat))   -> string
        |           : (string,nat)          -> char
        =           : (string,string)      -> bool
        /=          : (string,string)      -> bool
```

Spécification des séquences d'élément. Ce type est générique et exige la propriété *Formal_Sort* pour les éléments de la séquence.

```
Seq  seq
     nil  :                    -> seq[elem]
     <+   : (elem,seq[elem])   -> seq[elem]
```

Type abstrait des tableaux. Ce type est également générique et exige la propriété *Formal_Sort* pour ses éléments.

```
Array  array
       empty_array   : nat                    -> array[elem]
       store         : (array[elem],nat,elem) -> array[elem]
       |             : (array[elem],nat)      -> elem
       #             : array[elem]            -> nat
```

Type générique des ensembles.

```
Set  set
     empty_set        :                    -> set[elem]
     <+               : (elem,set[elem])   -> set[elem]
```

10.3.3- Les enrichissements

Spécification des opérateurs d'égalité et de différence entre les valeurs booléennes.

```
Eq_Bool  =    : (bool,bool) -> bool
         /=   : (bool,bool) -> bool
```

Définition de nouveaux opérateurs sur les valeurs d'entiers naturels.

```
Nat_Ops  /    : (nat,nat)   -> nat
         mod  : (nat,nat)   -> nat
         =    : (nat,nat)   -> bool
         /=   : (nat,nat)   -> bool
         <=   : (nat,nat)   -> bool
         <    : (nat,nat)   -> bool
         >=   : (nat,nat)   -> bool
         >    : (nat,nat)   -> bool
```

Définition de primitives génériques d'entrées-sorties.

```
Io   read     :        -> t
     write    : t      -> t
```

```
edit          : t        -> t
place_write   : (nat,t)  -> t
place_edit    : (nat,t)  -> t
```

Diverses définitions d'utilités générales.

```
Utilities   read_line  :        -> string
            jump_line  : nat    -> nat
            time       : bool   -> bool
            trace      : nat    -> nat
            place_tab  : nat    -> nat
            tab_val    :        -> nat
```

Enrichissement de la théorie des séquences avec d'autres opérateurs.

```
Seq_Ops  head       : seq[elem]              -> elem
         tail       : seq[elem]              -> seq[elem]
         +>         : (seq[elem],elem)       -> seq[elem]
         +          : (seq[elem],seq[elem])  -> seq[elem]
         #          : seq[elem]              -> nat
         is_empty   : seq[elem]              -> bool
```

Définition de l'égalité entre séquences d'éléments. Ce module est générique et utilise un opérateur d'égalité entre les éléments de la séquence.

```
Seq_Equality  =    : (seq[elem],seq[elem])  -> bool
              /=   : (seq[elem],seq[elem])  -> bool
```

Spécification de l'égalité entre tableaux.

```
Array_Equality  =  : (array[elem],array[elem]) -> bool
```

10.4- Modification des identificateurs OASIS

Nous avons vu dans le troisième chapitre de cette thèse que les identificateurs utilisés en OASIS se subdivisent en deux groupes en fonction des caractères employés.

Si le premier caractère est une majuscule, les reste de l'identificateur doit comporter des majuscules, blancs soulignés ou des chiffres. Si le premier caractère est un caractère minuscule ou spécial, les reste doit être formé de minuscules, blancs soulignés ou caractères spéciaux.

Cette contrainte a été jugée comme étant beaucoup trop forte. On aimerait, par exemple, manipuler l'identificateur *Seq* pour désigner une séquence. Une modification de OASIS a été réalisée afin de pouvoir admettre des identificateurs commençant par une majuscule, une minuscule ou un caractère spécial et qui contient, ensuite, des majuscules, minuscules, blanc soulignés, caractères spéciaux ou des chiffres.

Le diagramme syntaxique suivant résume les différentes possibilités.

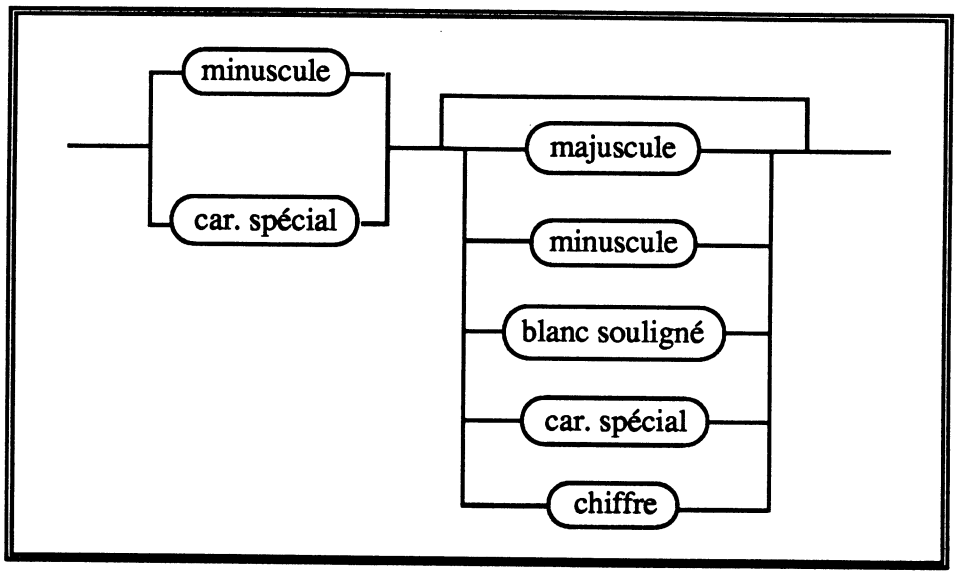


Diagramme syntaxique d'un identificateur de OASIS

Cette modification a porté sur un fichier source du système OASIS appelé *predext.pascal.*

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de

- Monsieur PAUL Etienne
- Monsieur JORRAND Philippe

Monsieur DRABIK Pascal

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "Informatique"

Fait à Grenoble, le 14 Novembre 1989

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
P. VENNEREAU

Résumé

La spécification par type abstrait algébrique permet de décrire le comportement de structures de données particulières telles que les files, piles, arbres etc. D'autre part, la généricité permet de spécifier le comportement de classes de structures.

Nous nous intéressons dans cet ouvrage aux types abstraits génériques structurés exprimés dans le formalisme algébrique. Ces spécifications génériques sont souvent liées entre elles par des relations telles que l'importation, la paramétrisation etc.

Toute instance d'une spécification générique S_1, paramétrée par une spécification S_2, doit vérifier des conditions décrites formellement dans S_2. En général, ce type de vérification se traduit par la démonstration de la validité de formules logiques dans une théorie particulière.

Nous définissons les différentes relations sémantiques qu'il faut valider dans le cadre d'un langage de programmation générique. Nous montrons également les principales caractéristiques de notre implantation. Celle-ci aide à réaliser les validations sémantiques en utilisant un démonstrateur semi-automatique de théorèmes.

Les expériences réalisées avec notre implantation nous ont conduit à constater le besoin de la réutilisation de démonstrations. Aussi avons nous proposé les fonctionnalités d'un atelier de démonstration en utilisant le formalisme algébrique.

Mots-clés

abstraction, type abstrait, spécification algébrique, programmation fonctionnelle, démonstrateur de théorème, généricité, paramétrisation, validation sémantique, théorie structurée.

Abstract

Abstract Data Types specifications allow the description of structures like queues, stacks, trees and so on. On the other hand, the genericity or parameterized specifications allows the description of the behaviour of classes of structures.

The aim of this thesis consists in the study of structured parameterized abstract data types in the algebraic framework. These generic specifications are often linked by relations such as importation, parameterization, etc.

Given a specification S_1, which is parameterized by another specification S_2, conditions described within S_2 must hold within each instantiation of S_1. In general, these verifications comes to prove the validity of logical formulae within a particular theory.

We define the semantic relations to be verified within the frame of a generic programming language. We describe then the main features of our implementation. This implementation brings help to a user in order to achieve semantic validations using mainly a theorem prover.

Finally, from the experiments lead with our implementation, we noticed the need of reusability of proofs. That is why we gave the outlines of a proof environment using the algebraic formalism.

Keywords

abstraction, abstract data type, algebraic specification, functional programming, theorem prover, genericity, parameterization, semantic validation, structured theory.