USTL
FLANDRES ARTOIS

LIFL
U.A 369 du C.N.R.S

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

N° d'ordre : 855

# THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

**DOCTEUR en INFORMATIQUE**

par

**PETITOT Michel**



## ALGEBRE NON COMMUTATIVE EN SCRATCHPAD : Application au problème de la réalisation minimale analytique

Thèse soutenue le **31 Janvier 1992** devant la commission d'Examen

**Membres du jury :**

| | |
|---|---|
| M. DAUCHET | Président |
| D. DUVAL | Rapporteur |
| M. FLIESS | Rapporteur |
| H. COMON | Examinateur |
| G. JACOB | Directeur de thèse |
| N.E. OUSSOUS | Examinateur |
| D. PINCHON | Examinateur |

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

---

## DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

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

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

## PROFESSEUR EMERITE

M. A. LEBRUN

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

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

## PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

## PROFESSEURS - CLASSE EXCEPTIONNELLE

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

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

## PROFESSEURS - 1ère CLASSE

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

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

## PROFESSEURS - 2ème CLASSE

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

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

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

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

*A ma mère,*
*qui femme de ménage dès l'âge de 14 ans,*
*a connu bien avant moi le difficile problème*
*de la réalisation minimale.*

# Remerciements

Quand je suis venu reprendre des études en Licence d'informatique, je pensais seulement obtenir un diplôme qui me permettrait d'enseigner la nouvelle option d'informatique en classe de seconde.

Je fus donc initié pendant une huitaine de jours au langage PASCAL par Jean-François Méhaut et Sophie Tison ; je ne sais pas pourquoi, mais ils m'ont donné envie de renouer avec le monde universitaire.

Je n'aurais peut-être pas choisi ce sujet de thèse où la partie implantation est conséquente si Jeannine et Bernard Leguy ne m'avaient pas communiqué leur passion de la programmation ; qu'ils en soient donc remerciés car je leur dois beaucoup.

Il ne m'est pas possible de remercier personnellement tous les enseignants en informatique du LIFL ; qu'ils sachent simplement que ce bouillon de culture m'a aidé à vivre.

Michel Mériaux, en tant que Directeur du Laboratoire, a la charge de veiller à ce que l'activité de recherche se passe dans de bonnes conditions matérielles et morales ; son bureau est largement ouvert à tous et je lui en sais gré.

Max Dauchet, Professeur au LIFL préside ce jury de thèse. Il a largement contribué à ma formation et a donc sa part dans ce travail.

Je remercie Hubert Comon, Chargé de Recherche CNRS à Université de Paris Sud, de me faire l'honneur de participer à ce jury.

Dominique Duval, Professeur à l'Université de Limoges, a eu la tâche ingrate de rapporter sur cette thèse. Son expérience et son engagement total pour développer les techniques de calcul formel sont pour moi une incitation à continuer mon travail de recherche.

Je remercie Didier Pinchon, Chargé de Recherche CNRS, d'avoir accepté de participer à ce jury et d'apporter ainsi son expérience dans l'implantation des algèbres de Lie.

J'exprime ma profonde gratitude à Christophe Reutenauer pour ses résultats théoriques qui sont à la base de ce travail.

Je suis ému de savoir que Michel Fliess, Directeur de Recherche CNRS au laboratoire des Signaux et Systèmes de Gif sur Yvette, ait accepté d'être rapporteur. Cette thèse tente modestement de rendre effectif son théorème sur la réalisation minimale analytique.

Tout au long de ce travail, j'ai bénéficié de l'appui constant de Nour-Eddine Oussous, Mître de Conférences au LIFL ; je le remercie de m'avoir dirigé tout en me laissant une grande liberté.

G. Jacob, Professeur au LIFL, Directeur de cette thèse, en est la cheville ouvrière et je pense qu'il sait d'instinct tout le respect que je lui porte.

Merci à Henri Glanc qui a réalisé avec soin la reproduction de cette thèse.

# Table des matières

# Index des notations

## Ensembles

| | |
|---|---|
| $X$ | Alphabet : ensemble totalement ordonné fini. |
| $X^*$ | Monoïde libre construit sur l'alphabet $X$. |
| $\mathcal{M}agma\langle X\rangle$ | Ensemble des mots complètement parenthésés. |
| $\mathcal{L}yndon\langle X\rangle$ | Base de Lyndon construite sur l'alphabet $X$. |
| $\mathcal{H}all\langle X\rangle$ | Base de Hall construite sur l'alphabet $X$. |
| $\mathcal{L}ie\langle X\rangle$ | Algèbre de Lie libre construite sur l'alphabet $X$. |
| $\mathcal{L}ie_{\leq n}\langle X\rangle$ | Ensemble des polynômes de Lie de degré $\leq n$. |
| $R\langle X\rangle$ | Polynômes non commutatifs construits sur l'anneau $R$. |
| $R\langle\langle X\rangle\rangle$ | Séries non commutatives construites sur l'anneau $R$. |
| $PBWL$ | Base de Poincaré–Birkoff–Witt associée à la base de Lyndon. |

## Autres notations

| | |
|---|---|
| $[l]$ | polynôme de Lie associé au mot de Lyndon $l$. |
| $Q_w$ | polynôme de la base $PBWL$ associé au mot $w$. |
| $S_w$ | élément de la base duale de la base $PBWL$. |
| $a \sqcup b$ | produit de mélange de deux mots, polynômes ou séries. |
| $ab$ | produit de Cauchy de deux mots, polynômes ou séries. |
| $< S \mid w >$ | coefficient du mot $w$ dans la série $S$. |
| $S \triangleright w$ | résiduel à droite de $S$ par le mot $w$. |
| $w \triangleleft S$ | résiduel à gauche de $S$ par le mot $w$. |
| $(\Sigma)$ | système dynamique analytique. |
| $h$ | fonction d'observation du système $(\Sigma)$. |
| $Y_i$ | $i^{\text{ième}}$ champ de vecteurs d'un système $(\Sigma)$. |
| $u_i(t)$ | $i^{\text{ième}}$ entrée d'un système $(\Sigma)$. |
| $Y_i \circ h_{\mid q}$ | évaluation de la fonction $Y_i \circ h$ en $q$. |
| $\mathcal{Y}(w)$ | opérateur différentiel obtenu par composition des $Y_i$. |
| $\mathcal{E}_u(w)$ | intégrale itérée sur le mot $w$ pour l'entrée $u$. |
| $\mathcal{H}_u(t)$ | opérateur de transport associé à l'entrée $u$. |
| $S_u(t)$ | série de Chen associée à une entrée $u$. |

# 0

# Introduction générale

Le problème de la réalisation minimale analytique fait partie des "classiques" de l'automatique. Il consiste à retrouver les équations différentielles (de Kalmann) qui gouvernent l'évolution d'un système dynamique analytique à partir de son comportement Entrée/Sortie.

Nous donnons l'implantation d'un algorithme efficace pour calculer une représentation d'état minimale d'un système dynamique analytique dont le comportement Entrée/Sortie est donné par sa série génératrice lorsque celle-ci est polynomiale.

Ceci constitue par la-même une preuve constructive de la conjecture de G.Jacob restée indémontrée:

> *Une série génératrice polynomiale admet une réalisation minimale polynomiale (la fonction d'observation et les composantes des champs de vecteurs sont des polynômes).*

On sait d'après M.Fliess [9] que la réalisation minimale analytique est unique à un changement de coordonnées près (difféomorphisme analytique) dans l'espace d'état.

La réalisation que nous calculons est canonique au sens où elle ne dépend que de l'ordre choisi pour comparer les lettres de l'alphabet de codage. Ce résultat est acquis en utilisant les techniques de "base-standard" appliquées aux espaces vectoriels de polynômes.

De nombreux chercheurs ont travaillé sur la réalisation minimale analytique et les problèmes qui lui sont liés comme l'observabilité et la contrôlabilité en automatique non linéaire. Citons entre autres R.Hermann et A.Krenner(1977) [11], J.H.Sussman(1977) [32], B.Jakubczyk(1980) [16] et M.Fliess(1983) [9].

A partir des travaux de M.Fliess, C.Reutenauer(1986) dans [30] a donné une version très "syntaxique" de l'existence de la réalisation minimale en effectuant son calcul dans une certaine algèbre de mélange. En partant d'une série polynomiale, la réalisation qu'il obtient est analytique (mais pas forcément polynomiale) et la traduction informatique de sa preuve ne donne pas un algorithme qui termine en un nombre fini de pas.

Utilisant les résultats de C.Reutenauer, NE. Oussous dans [24, 26, 25] a implanté, dans le système de calcul formel Macsyma, un algorithme qui, par une méthode d'exploration systématique de toutes les solutions possibles[1], calcule une réalisation minimale polynomiale mais sans toutefois démontrer que celle-ci existe dans tous les cas.

L'algorithme que nous proposons, outre qu'il est prouvé, est beaucoup plus rapide. Ainsi,

[1] La méthode relève de l'algèbre linéaire

une série génératrice de 3 variables en degré 7 est réalisée en moins d'une minute sur un PC-RT ©IBM avec le système de calcul formel SCRATCHPAD (voir la démonstration page 143).

Le travail d'implantation doit être considéré comme un travail de recherche, d'abord parce qu'il conduit à se poser des problèmes non triviaux de représentation des objets. Il est clair que certains concepts traditionnels des mathématiciens (nombres réels, fonctions analytiques, séries formelles) ne sont pas effectifs car ces ensembles ne sont pas dénombrables et donc leurs éléments ne peuvent être représentés par un nombre fini de symboles. Il reste alors à combler le vide lorsqu'on a pris conscience de cette difficulté.

Quand on se restreint à des domaines tels que les polynômes, le problème de la représentation demeure crucial si l'on désire obtenir de bonnes performances au niveau du temps de calcul et de la taille des données. S'agissant des polynômes en variables non commutatives, le lecteur constatera qu'il y a au moins quatre façons de les représenter et que chacune d'elle a son utilité.

Les techniques modernes de "génie logiciel" visent à la réutilisabilité des composants logiciels. Le système de calcul formel SCRATCHPAD basé sur la théorie des types abstraits et la généricité s'inscrit dans ce courant. On devra donc considérer les divers modules présentés, non comme une collection d'algorithmes plus ou moins subtils, mais comme une tentative de structurer la partie "algèbre non commutative" d'un système de calcul formel.

Le premier chapitre est une initiation au système de calcul formel SCRATCHPAD. Il pourra paraître assez indigeste à certains ; sa lecture n'est nullement indispensable pour comprendre le reste de l'exposé. D'une façon générale, j'aurais pu mettre complètement à part le code SCRATCHPAD qui, pour être plus lisible que du FORTRAN, produit cependant une coupure dans le texte en français. Seules les spécifications (mode d'emploi) des principaux domaines figurent. Autant que faire se peut, ce code est assorti de commentaires rédigés en "clair".

Le deuxième chapitre introduit quelques opérations et notations de base en algèbre non commutative. La partie implantation des polynômes sous forme distribuée et récursive est essentielle d'un point de vue pratique, car elle conditionne l'efficacité de l'algorithme de la réalisation minimale.

Le troisième chapitre traite de l'algèbre de Lie libre et de la base de Poincaré-Birkoff-Witt pour l'algèbre enveloppante. Les algorithmes présentés sont essentiellement tirés des travaux de G. Melançon et C. Reutenauer[21]. On trouvera une implantation des polynômes de Lie dans la base de Lyndon en partant d'un principe simple:

*Les équations de définition* [2] *d'une algèbre de Lie*

$$\begin{cases} [a,a] = 0 \\ [a,b] = -[b,a] \\ [[a,b],c]+[[b,c],a]+[[c,a],b] = 0 \end{cases}$$

*permettent de générer un système de réécriture confluent noethérien*

$$\begin{cases} [a,a] \longrightarrow 0 \\ [a,b] \longrightarrow -[b,a] \quad si\ a > b \\ [[a,b],c] \longrightarrow [a,[b,c]]+[[a,c],b] \quad si\ b < c \end{cases}$$

[2] Je n'ai pas fait figurer les formules exprimant la bilinéarité du crochet de Lie

*l'ordre sur les crochets étant l'ordre lexicographique calculé sur leur forme déparenthésée.*

Pour ce système de réécriture, la forme normale d'un polynôme de Lie quelconque est alors sa décomposition dans la base de Lyndon.

Le théorème de Poincaré-Birkoff-Witt nous fournit une nouvelle représentation possible des polynômes non commutatifs. Contrairement à toute attente, les résultats, au niveau du temps de calcul, font que cette implantation est largement utilisable. Dans certains cas, ils sont même meilleurs que pour l'implantation récursive ; ceci est dû au fait que le nombre de mots figurant dans la représentation d'un polynôme est quelquefois plus faible dans la base de Poincaré-Birkoff-Witt (voir fichier de démonstration en annexe page 137). On trouvera également exposés tous les algorithmes permettant de changer de base.

Le chapitre IV sur les bases-standard ne prétend pas apporter d'éléments nouveaux dans ce domaine de la réécriture algébrique mais fournit un cadre théorique assez facilement transposable pour l'étude des idéaux en algèbre différentielle ou dans l'algèbre de Weyl. L'accent est mis sur le fait qu'un système de réécriture n'est pas seulement la donnée d' un ensemble de règles ; il convient également de bien préciser de quelle manière on effectue les substitutions, ce que d'aucuns appellent le *filtrage*.

L'usage qui est fait ici de l'algorithme de Knuth-Bendix se limite à l'étude des espaces vectoriels de polynômes et des idéaux à droite finiment engendrés. Dans les deux cas, la théorie mathématique en est très élémentaire.

La motivation est essentiellement pratique ; les calculs de dépendance linéaire sont fréquents en calcul formel. Il se trouve que l'implantation des espaces vectoriels, fournie en standard avec SCRATCHPAD , basée sur le calcul matriciel, est parfaitement inefficace en algèbre non commutative. L'algorithme de la réalisation minimale reprogrammé en utilisant les techniques de bases-standard est nettement plus rapide.

Le calcul de bases-standard pour les idéaux de polynômes non commutatifs découle d'une volonté d'implanter efficacement les séries rationnelles. Ces séries bien étudiées mathématiquement sont souvent définies par une représentation matricielle. Cette représentation, même lorsqu'elle est minimale n'est pas unique. On montre qu'il est possible de représenter canoniquement une série rationnelle par la donnée d'un polynôme et de la base-standard de son annulateur à droite (resp. idéal syntaxique) qui est un idéal à droite (resp. idéal bilatère). Il est trop tôt pour conclure si cette idée est bonne puisqu'une telle implantation des séries rationnelles n'est pas encore réalisée.

Le chapitre V sur les algèbres de mélange est destiné à préparer l'exposé, au chapitre suivant, de l'algorithme de calcul de la réalisation minimale . Il est complètement basé sur des résultats de C.Reutenauer [30]. Du point de vue du Calcul formel, les bases de l'algèbre de mélange fournissent une nouvelle représentation possible des polynômes en variables non commutatives ; l'intérêt majeur réside dans le fait que la taille des polynômes n'explose plus quand on calcule des produits de mélange. Je compte donc réimplanter l'algorithme de la réalisation minimale (ce sera la version 5) en représentant la série génératrice dans la base duale de la base de Poincaré-Birkoff-Witt associée à la base de Lyndon.

Le résultat théorique essentiel se trouve exposé au chapitre VI ; on y démontre qu'*une série génératrice polynomiale admet une réalisation minimale polynomiale.* J'ai essayé de rédiger la démonstration mathématique en prouvant un algorithme qui répond à la question posée.

Il se trouve, qu'en plus, l'algorithme est assez rapide d'exécution. Ce résultat s'inscrit dans les nombreuses recherches en algèbre, pour rendre effectifs des résultats basés sur la théorie des fonctions analytiques et le théorème des fonctions implicites.

Cette thèse se termine par une implantation du groupe de Lie d'une algèbre nilpotente, assurant ainsi la maîtrise complète des exponentielles de Lie. On ne trouvera rien de nouveau, du point de vue théorique, sur ce sujet étudié depuis de longues années par les mathématiciens. L'exposé vise simplement à montrer qu'un certain savoir-faire en calcul formel est succeptible de résoudre partiellement le difficile problème du "motion-planning" en français: commande exacte sur état cible. Ce problème sur lequel travaille J.H.Sussmann [19, 33] a été récemment repris par G.Jacob [13], directeur de cette thèse.

# 1

# Pour comprendre Scratchpad

Résumé:

- SCRATCHPAD est le seul système de calcul formel typé; il utilise massivement la *généricité*; cette généricité est contrôlée par la notion de *catégorie*.
- Le typage des données est statique et implicite.
- SCRATCHPAD s'inspire de la programmation fonctionnelle.

Avertissement:

J'utilise la version SCRATCHPAD 2.2 du 15/11/89. Il se peut donc que certaines affirmations présentes dans ce chapitre ne soient pas valables pour les versions ultérieures.

## 1.1 Bref historique

SCRATCHPAD est né à partir des années 70, conçu par un petit groupe travaillant au:

```
Computer Algebra Group, Mathematical Sciences Department
IBM Thomas J. Watson Research Center, Box 218
Yorktown Heights, New York  10598  USA
```

Les systèmes de l'époque étaient tous très influencés par la programmation fonctionnelle, en particulier par l'esprit du LISP.

Lisp permet d'implanter assez facilement les briques de base d'un système de calcul formel:

- manipulation d'entiers arbitrairement grands
- manipulation des polynômes, fractions rationnelles et plus généralement des expressions mathématiques représentées sous forme arborescente.

Il est un moyen efficace pour implanter des systèmes de réécriture.
**Exemple:** règles de dérivation:

$$\begin{array}{lcl} (u+v)' & \rightarrow & u'+v' \\ (uv)' & \rightarrow & u'v+uv' \\ (cte)' & \rightarrow & 0 \end{array}$$

L'inconvénient majeur des programmes écrits directement en LISP est que les objets manipulés ne sont pas typés; la sémantique des programmes est donc peu claire et les erreurs à l'exécution difficiles à identifier.

En revanche, les langages typés des années 70 (Pascal, C, PL1) permettaient difficilement le calcul symbolique.

La solution est progressivement apparue avec la théorie des types abstraits qui a, entre autre, débouchée sur le langage ADA et l'utilisation intensive de la généricité. Nous verrons que SCRATCHPAD va beaucoup plus loin dans ce domaine puisque la généricité y est contrôlée par la notion de catégorie.

Le système SCRATCHPAD est désormais commercialisé par la société NAG sous le nom de AXIOM.

## 1.2 Notion de type en général

Typer un objet consiste à définir trois choses :

- L'ensemble de ses valeurs possibles.
- La façon dont cet objet est représenté en mémoire.
- L'ensemble des opérations permises sur cet objet.

Tout objet typé possède alors :

- un nom,
- une valeur,
- un type.

L'utilisateur définit ses propres types à partir des *types de base* en utilisant des *constructeurs de type.*

Exemple en Turbo–Pascal :

```
type VECTEUR = array[1..100] of Integer;
var x:Integer;
    v:VECTEUR;
begin
  x:=3;
  .....
end.
```

Commentaires :

- **x** est le nom d'une variable de type **Integer**.

- `Integer` est un *type de base* représenté en mémoire par 16 bits et comportant un ensemble de valeurs allant de $-2^{15}$ à $2^{15}-1$.
- `3` est un *littéral* auquel le compilateur attribue le type `Entier_universel` compatible avec le type `Integer`; ceci permet au compilateur d'effectuer un contrôle de type lors de l'affectation `x:=3`.
- `VECTEUR` est un type défini par l'utilisateur en utilisant le *constructeur de type* `array`.

## 1.3 Le typage en SCRATCHPAD

### 1.3.1 Les types de base

Les types de base en SCRATCHPAD sont : `Float`, `Integer`, `String`, `Boolean` et `Symbol`.

Les objets appartenant aux types de base peuvent être référencés par des littéraux.

Exemples:

- `"Bonjour !!"` est un littéral de type `String`,
- `true` est un littéral de type `Boolean`.

### 1.3.2 Les objets

Les objets sont des entités créées et manipulées par des fonctions. Il y a quatre sortes d'objets :

- Objets calculés,
- Fonctions,
- Domaines,
- Catégories.

En SCRATCHPAD tous les objets peuvent être affectés ou passés en paramètre.

```
ENTIER := Integer        -- affectation d'un domaine

   (1)  Integer
                                                        Type: Domain

double(x:ENTIER):ENTIER == 2*x
   Function declaration double : Integer -> Integer has been added to
   workspace.
                                                          Type: Void
double                   -- double est le corps d'une fonction
```

```
   (3)  double x == 2x
                                                    Type: FunctionCalled double


f: ENTIER -> ENTIER      -- definition de la signature de la fonction f
                                                                  Type: Void

f := double              -- f est une fonction compilee
   Compiling function double with type Integer -> Integer

   (4)  theMap(*1double;1,655)
                                                    Type: (Integer -> Integer)

f(3)

   (5)  6
                                                              Type: PositiveInteger
g:=f                     -- affectation d'une fonction

   (6)  theMap(*1double;1,655)
                                                    Type: (Integer -> Integer)
```

### 1.3.3 Les fonctions

SCRATCHPAD ignore la notion de procédure. Il utilise les fonctions à la manière du langage C :

- les paramètres des fonctions sont transmis par valeur,
- le résultat d'une fonction n'est pas forcément affecté à une variable,
- une fonction peut renvoyer un résultat de type **Void**,
- tout bloc de programme est considéré comme une fonction qui renvoie une valeur dont le type est calculé à la compilation.

### 1.3.4 Le type des objets

1. Le type d'un objet calculé est son domaine.
2. Le type d'une fonction est sa signature.
3. Le type d'un domaine est **Domain**.
4. Le type d'une catégorie est **Category**.

### 1.3.5 Les unités de compilation

Les unités de compilation sont de trois sortes :

- les constructeurs de domaine,
- les constructeurs de catégorie,
- les paquetages.

Chaque unité de compilation a un nom complet ainsi qu'un nom abrégé. Contrairement à ADA, les résultats de compilation de deux unités distinctes sont stockés dans des librairies distinctes.

#### 1.3.5.1 Exemple: les matrices carrées

La commande `)show SM` de l'interpréteur affiche à peu près ceci: [1]

```
M[ndim: PI]R: RING  is a domain constructor
Abbreviation for SquareMatrix is SM
This constructor is currently exposed.
Issue )edit /spad/nalgebra/matrix.spad to see algebra source code for SM

------------------------------- Operations -------------------------------
@*@ : (R,$) -> $                    @*@ : ($,$) -> $
@*@ : (I,$) -> $                    @**@ : ($,NNI) -> $
@+@ : ($,$) -> $                    @-@ : ($,$) -> $
-@ : $ -> $                         @/@ : ($,R) -> $
@=@ : ($,$) -> B                    EchelonLastRow : $ -> $
1 : () -> $                         0 : () -> $
characteristic : () -> NNI          coerce : V R -> $
.......
```

`SquareMatrix` est un *constructeur de domaine* permettant de gérer les matrices carrées de taille quelconque `ndim` et contenant des éléments dont le type est un domaine `R` appartenant à la catégorie des anneaux.

Comme en ADA, les unités forment un ensemble strictement hiérarchisé. Ainsi une unité de compilation utilise d'autres unités de compilation appelées ses ancêtres et peut à son tour être utilisée par d'autres que nous appelerons ses dépendances.
Contrairement à d'autres langages comme C, ce système strictement hiérarchisé interdit des appels croisés: il est impossible que l'unité A utilise l'unité B alors que l'unité B utilise l'unité A.

### 1.3.6 Typage implicite

Tout objet en SCRATCHPAD a un type mais ce type n'est pas forcément déclaré par l'utilisateur. Le type d'une variable peut être inféré par le compilateur ou l'interpréteur de SCRATCHPAD lors de sa première affectation.

```
(2) ->x:=2
```

[1] Pour des raisons liées au logiciel Tex, le caractère □ est ici remplaçé par @

```
(2)  2

Type: PositiveInteger
```

La variable $x$ a donc pour valeur 2 et pour type **PositiveInteger**.

Remarquons que le typage est ici *implicite*: le type de $x$ n'a pas été déclaré par l'utilisateur mais a été inféré par l'interpréteur de SCRATCHPAD. L'inférence comporte une part d'arbitraire; 2 pourrait aussi bien être typé **Integer** ou **NonNegativeInteger**. On aurait pu évidemment déclarer explicitement le type de $x$:

```
(2) ->x:I:=2

(2)  2

Type: Integer
```

En mode compilé, le type d'une variable doit être explicitement déclaré sauf si

1. La variable balaye un ensemble de valeurs dont le type ne présente aucune ambiguité.

```
for i in 1..100 repeat .....
```

2. La variable est en partie gauche d'une affectation dont la partie droite a un type ne présentant aucune ambiguité.

```
x:=true
```

En principe, le compilateur vérifie que toutes les variables sont initialisées, ce qui n'est pas toujours un problème simple !!.

**Remarque**: Le compilateur de SCRATCHPAD II n'a pas détecté l'erreur de typage de la variable **x** dans la fonction suivante:

```
essai(i:Integer):String ==
  if i > 10 then x:=99
            else x:=true
  "fait"
```

En fait pour SCRATCHPAD , ce n'est pas une errreur; le type de **x** sera inféré comme étant **Union(Integer,Boolean)**.

### 1.3.7 L'assemblage des constructeurs de domaines

Un des avantages de SCRATCHPAD est de permettre la définition de domaines extrêmement complexes. Nous verrons que la cohérence des constructions effectuées est assurée grâce à la notion de catégorie.

#### 1.3.7.1 Exemple

```
(4) ->  MatriceDePolynomes := SquareMatrix(2,Polynomial(Integer))

   (4)  M[2]P I

   Type: DOMAIN

(5) ->  mp:MatriceDePolynomes :=[[x+3,x-1],[0,1]]
```

$$(5) \qquad \begin{bmatrix} x+3 & x-1 \\ 0 & 1 \end{bmatrix}$$

```
   Type: M[2]P I

(6) ->  mp2:=mp**2
```

$$(6) \qquad \begin{bmatrix} x^2+6\,x+9 & x^2+3\,x-4 \\ 0 & 1 \end{bmatrix}$$

```
   Type: M[2]P I
```

La cohérence de la construction (4) est assurée parce que les domaines **Polynomial(Integer)** et **Integer** appartiennent à la catégorie des anneaux. Cette cohérence est vérifiée automatiquement par SCRATCHPAD.

Il est évident que la construction de domaines en utilisant d'autres domaines n'est possible que si l'on dispose au départ de domaines prédéfinis tels que **Integer, Boolean** etc ...
Ces types prédéfinis sont implantés en LISP.

### 1.3.8 Conversions de type

#### 1.3.8.1 Fonctionnement en mode interprété

Remarquons que l'interpréteur de SCRATCHPAD effectue des conversions de type non demandées explicitement par l'utilisateur. Dans la pratique, ces conversions peuvent prendre beaucoup plus de temps que les calculs proprement dits, ainsi en (5)

- la variable $x$ est convertie en **Expression**, puis en **SortedExpression**, puis en polynôme à coefficients entiers,
- les entiers 0,1,3 sont convertis en polynômes,
- les opérations d'addition, soustraction de polynômes sont alors effectuées; on obtient une liste de polynômes,

- la liste est convertie en **MatriceDePolynomes**.

Toutes ces conversions ne peuvent s'effectuer que parce qu'elles ont été prévues et programmées dans les diverses unités de compilation sous le nom conventionnel **coerce**. L'interpréteur doit rechercher dans les diverses unités de compilation une chaîne de conversions compatible avec les objets dont le type est parfaitement déterminé. On conçoit que l'algorithme de recherche des bonnes conversions puisse prendre beaucoup de temps.

#### 1.3.8.2 Les conversions demandées explicitement

Il existe un opérateur de conversion :: dont l'effet est identique à l'appel d'une fonction **coerce**. Ces conversions qui sont des fonctions comme les autres, apportent beaucoup de souplesse dans les calculs.
**Exemple:** une matrice de polynômes est transformée en polynôme à coefficients matriciels

```
(7) ->mp2::P SM(2,I)
```

$$(7) \qquad \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} x^2 + \begin{bmatrix} 6 & 3 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 9 & -4 \\ 0 & 1 \end{bmatrix}$$

```
Type: P M[2]I
```

#### 1.3.8.3 L'indépendance entre les représentations interne et externe

Chaque fois que l'interpréteur de SCRATCHPAD doit afficher un objet de type **$**, il exécute une conversion de type **$ --> Expression** puis l'expression obtenue est affichée grâce à une primitive prédéfinie **mathprint** écrite en LISP. Toute unité de compilation définissant un constructeur de domaine doit obligatoirement comporter une fonction **coerce** du type **$** vers le type **Expression**.

## 1.4 La généricité

### 1.4.1 Idée 1

*Implémenter les algorithmes à leur niveau naturel de généralité.*

#### 1.4.1.1 Exemple

```
pgcd(a,b) == if b=0 then a else pgcd(b,remainder(a,b))
```

Cet algorithme doit pouvoir fonctionner en toute généralité sur n'importe quel domaine appartenant à la catégorie des anneaux et sur lequel existe une division euclidienne. Nous verrons que SCRATCHPAD permet de définir une catégorie telle que **EuclideanDomain**.

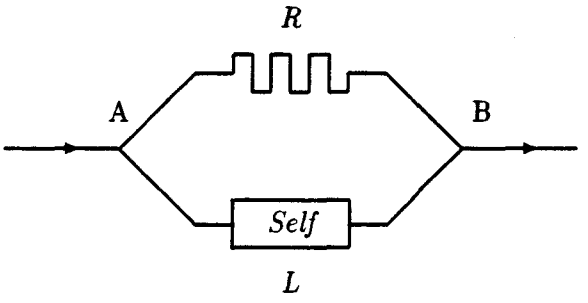
Notons que l'écriture de procédures génériques suppose que le nommage des fonctions soit harmonisé; ainsi, si le reste de la division de deux entiers $a$ et $b$ est noté `remainder(a,b)`, il faut respecter cette notation pour les autres anneaux enclidiens.

### 1.4.2 Idée 2

*La généricité permet de traiter de manière typée des problèmes relevant du calcul symbolique et cela en utilisant des domaines tels que les polynômes, fractions rationnelles, espaces de fonctions etc...*

#### 1.4.2.1 Exemple

Soit à effectuer des calculs sur les impédances complexes dans un circuit électrique alternatif.




L'impédance du tronçon AB est égale à

$$\frac{RLi}{R+Li} = \frac{RL^2+R^2Li}{R^2+L^2} = \frac{RL^2}{R^2+L^2} + i\frac{R^2L}{R^2+L^2}$$

Le résultat est donc un nombre complexe dont les parties réelle et imaginaire sont des fractions rationnelles en $R$ et $L$.
Le type utilisé en SCRATCHPAD peut être `Gaussian RationalFunction Integer`.

Il est donc essentiel de disposer d'un constructeur de domaine `Gaussian` générique paramétré par un anneau quelconque; ici l'anneau utilisé est l'anneau des fractions rationnelles à coefficients entiers.

### 1.4.3 Un abandon du calcul symbolique ?

Les systèmes de calcul formel non typés ont pour noyau un simplificateur universel, en général bien adapté à l'algèbre commutative, permettant de réduire autant que faire se peut une expression symbolique quelconque.

SCRATCHPAD ne possède pas de simplificateur car, d'une certaine façon, il ne manipule pas des symboles mais des objets typés qui doivent **posséder une valeur avant d'être utilisés:**

```
(1) ->i:Integer
                                                    Type: Void
(2) ->x:Integer := 2*i
    i is declared as being in Integer but has not been given a value.
```

Cette façon de procéder a, dans certains cas, un inconvénient majeur. Il n'est pas facile de faire le calcul de $(x^n+1)(x^n-1)$ pour obtenir $x^{2n}-1$. On aurait envie de considérer ce calcul comme un produit de 2 polynômes en $x$ où l'exposant $n$ désigne un **symbole** entier. SCRATCHPAD pour typer les expressions fera appel à un domaine du genre `FunctionalExpression Integer` qui nécessitera le chargement en mémoire d'une grande quantité de bibliothèques dont rien ne dit qu'elles tiendront facilement en mémoire.

**Cette difficulté est incontournable sans l'ajout à SCRATCHPAD d'une nouvelle notion lui permettant de gérer les symboles typés.**

## 1.5 La notion de catégorie

### 1.5.1 Qu'est-ce qu'une catégorie ?

*Une catégorie est la donnée de deux ensembles*

1. *un ensemble de fonctions exportées, caractérisées par leur signature*
2. *un ensemble d'attributs décrivant la sémantique de ces fonctions*

*Dans les dernières versions de* SCRATCHPAD *la définition d'une catégorie peut éventuellement comporter une implantation par défaut de certaines fonctions.*

**Exemple** : voici comment est définie en SCRATCHPAD la catégorie des ensembles ordonnés:

```
)abbrev category ORDSET OrderedSet

OrderedSet(): Category == Set with
  --operations
    "<": ($,$) -> Boolean
    max: ($,$) -> $
    min: ($,$) -> $
  --attributes
    irreflexive("<")              --not (x < x)
    transitive("<")               --x < y and y < z  implies  x < z
    total("<")                    --not(x < y) and not(y < x)  implies x=y
  add
  --declarations
    x,y: $
  --definitions
    max(x,y) ==
      x > y => x
      y
    min(x,y) ==
      x > y => y
      x
```

La commande `)show` de l'interpréteur affiche alors:

```
(8) ->)sh ORDSET
 OrderedSet is a category constructor.
 Abbreviation for OrderedSet is ORDSET
 This constructor is currently exposed.
 Issue )edit /spad/nalgebra/catdef.spad to see algebra source code for ORDSET

------------------------------- Operations -------------------------------

 @<@ : ($,$) -> B                    @=@ : ($,$) -> B
 coerce : $ -> E                     coerce : E -> Union($,"failed")
 max : ($,$) -> $                    min : ($,$) -> $


------------------------------- Attributes -------------------------------

  irreflexive(<)                      total(<)
  transitive(<)
```

### 1.5.2 Réunion de 2 catégories

Cela consiste à définir une nouvelle catégorie obtenue en faisant l'union de l'ensemble des fonctions exportées et des attributs figurant dans les 2 catégories de départ [2].

La réunion de deux catégories est réalisée par le mot réservé **Join** et l'ajout de fonctions ou d'attributs par le mot **with**. Voici par exemple comment est définie la catégorie des monoïdes abéliens ordonnés:

```
OrderedAbelianMonoid(): Category == Join(OrderedSet,AbelianMonoid) with
    --attribute
      orderPreserve("+")                --if x < y %then x+z < y+z
```

### 1.5.3 Comparaison avec la programmation orientée objet

Sur les exemples, nous constatons que la notion de catégorie correspond à la notion de **classe abstraite** dans les langages orientés objet. Le connecteur **Join** réalise un héritage multiple quant aux propriétés d'un objet. Ainsi la classe des **Integer** hérite des propriétés des ensembles ordonnés et des anneaux.
Il est possible d'interroger la base de données de SCRATCHPAD:

```
(8) ->  I has ORDSET

   (8)  true

   Type: Boolean
```

[2] Un conflit peut se produire sur les implantations par défaut (problème de l'héritage multiple dans les Langages orientés objet)

```
(9) -> I has Ring

   (9)  true

   Type: Boolean

(10) -> I has orderPreserve("+")

   (10)  true

    Type: Boolean
```

Il est possible (voir page 31) d'exprimer grâce aux attributs certaines propriétés de transfert telles que:

> *Si un certain anneau R n'a pas de diviseurs de zéro, l'anneau des polynômes à coefficients dans R n'a pas de diviseurs de zéro.*

SCRATCHPAD est alors capable de calculer la valeur de ces attributs:

```
(11) -> matrice:= SM(2,I)

   (11)  SquareMatrix(2,Integer)
         Type: Domain

(12) -> P matrice has noZeroDivs     -- polynomes a coefficients matriciels

   (12)  false
         Type: Boolean

(13) -> P I has noZeroDivs           -- polynomes a coefficients entiers

   (13)  true
         Type: Boolean
```

### 1.5.4 Limitations

On pourra déplorer le manque de souplesse quant au choix des identificateurs pour les fonctions à implanter. Ainsi dans la catégorie des **groupes**, la loi de composition est notée * et donc l'ensemble des **Integer** muni de l'addition notée + n'appartient pas à la catégorie des **groupes**.

D'une façon plus générale, un domaine appartenant à la catégorie des **groupes abéliens** ne peut pas appartenir à la catégorie des **groupes** car dans un cas, la loi de composition est notée + et dans l'autre *.

```
(10) ->AbelianGroup has Group

   (10)  false
```

```
Type: B
```

Nous constatons donc sur cet exemple que la notion de catégorie en SCRATCHPAD ne recoupe pas complètement la notion de catégorie en Mathématiques; il est en effet impossible d'exprimer en SCRATCHPAD l'idée que

1. $(\mathbb{Q}, +, 0)$ est un **groupe**
2. $(\mathbb{Q} \setminus \{0\}, \times, 1)$ est un autre **groupe**

## 1.6 Paquetages et constructeurs de domaines

### 1.6.1 Les constructeurs de domaine

#### 1.6.1.1 Structure de base

C'est une unité de compilation permettant de définir un nouveau domaine; celui-ci est référencé à l'intérieur de l'unité par `$`. Un constructeur de domaine comporte obligatoirement deux parties:

1. La catégorie (*partie publique*) contenant les spécifications:
   - Signatures des opérations et fonctions exportées
   - Attributs précisant la sémantique de ces opérations
2. La capsule (*partie privée*) contenant:
   - La description de la représentation interne `Rep` des objets du domaine `$`
   - L'implantation des différents algorithmes de calcul

#### 1.6.1.2 Exemple

Soit à gérer l'ensemble des mots construits sur un alphabet `S` donné. On désire reprendre l'implantation déjà existente de `FreeMonoid(S)` en la complètant par quelques fonctionnalités supplémentaires, en particulier une définition d'un ordre sur les mots.

```
)abbrev domain OFMON OrderedFreeMonoid

OrderedFreeMonoid(S: OrderedSet): OFMcategory == OFMdefinition where
    NNI ==> NonNegativeInteger

    OFMcategory == OrderedMonoid with
        first: $ -> S                    -- 1 ere lettre d'un mot
        rest:  $ -> $
        lexico: ($,$) -> Boolean         -- ordre lexicographique
```

```
.....

OFMdefinition == FreeMonoid(S) add
  -- representation
    Term := Record(gen: S, exp:Integer )
    Rep:= List Term
  -- definitions
    x < y ==                    -- ordre lexicographique par longueur
       lx:NNI := length x
       ly:NNI := length y
       lx = ly => lexico(x,y)
       lx < ly

    ......
```

#### 1.6.1.3 Remarque destinée aux adeptes de la programmation orientée objet

Sur cet exemple, nous constatons qu'il y a lieu de distinguer deux notions possibles sur l'héritage:

- héritage de spécifications: l'ensemble construit est un `OrderedMonoid`. **Cet héritage peut être multiple** lorsqu'on utilise le connecteur `Join` réalisant la réunion de deux catégories (voir 1.5.2 page 17).
- héritage d'une implantation: la gestion des mots dans `OrderedFreeMonoid1(S)` hérite de l'implantation des fonctions de `FreeMonoid(S)` ainsi que de la représentation interne `Rep` des objets de ce domaine (les mots sont représentés par une liste de couples (lettre,exposant)). **Cet héritage est nécessairement un héritage simple.**

  Lorsque le type `Rep` est hérité, il n'est pas obligatoire de le redéfinir, sauf si l'on désire manipuler directement la représentation interne des objets [3] ; dans ce cas SCRATCHPAD vérifie que les deux représentations sont des types *isomorphes*, notion que nous reverrons plus tard.

#### 1.6.1.4 Conflits entre Rep et $

Les fonctions exportées par le domaine `Rep` peuvent être en conflit de nom avec les fonctions exportées par le domaine `$`. Dans l'exemple précédent il y a conflit entre

1. la fonction `first` du domaine `Rep` qui renvoie le premier terme de la liste représentant un mot

   $$first(a^3b) \text{ renvoie } (a,3)$$

2. la fonction `first` du domaine `$` qui renvoie la première lettre d'un mot

   $$first(a^3b) \text{ renvoie } (a)$$

Il y a lieu de lever toute ambiguité en utilisant les notations `first$$` et `first$Rep`.

[3] SCRATCHPAD donne donc la possibilité de "violer" le principe d'encapsulation des données

### 1.6.2 Les paquetages

Contrairement à un *constructeur de domaine*, un **package** ne définit pas une nouvelle classe d'objets. C'est une unité de compilation contenant l'implantation d'un certain nombre de fonctions que l'on a voulu regrouper pour une raison quelconque (voir 2.2.7.1).

Cependant sa structure de base est identique à celle d'un constructeur de domaine à savoir:

- Une catégorie
- Une capsule

Un **package** peut comporter des paramètres génériques. Naturellement, **Rep** ne figure pas dans un paquetage et le symbole **$** ne peut être employé.

Dans les dernières versions de SCRATCHPAD, les paquetages sont pratiquement inutilisés. La distinction entre paquetages et constructeurs de domaine n'est pas indispensable mais elle apporte une information supplémentaire permettant à un programmeur de comprendre une implantation réalisée par d'autres. Elle permet d'alléger le contenu des constructeurs de domaine; ceux-ci ne comportent plus que les fonctions de base *(primitives)* indispensables pour manipuler efficacement les objets en faisant abstraction de leur représentation interne (concept d'*encapsulation* des données). Certains algorithmes dont l'implantation se fait facilement à partir des primitives de calcul peuvent alors être déportés dans des paquetages.

## 1.7 SCRATCHPAD et la programmation fonctionnelle

SCRATCHPAD a beaucoup emprunté à la programmation fonctionnelle et ses techniques de compilation. En particulier les fonctions y prennent une grande place et sont des objets que l'on peut affecter ou passer en paramètre.

### 1.7.1 Tout (ou presque) est fonction

Les constructeurs de domaine ou de catégorie sont considérés comme des fonctions et leur définition (voir [10]) obéit à la même syntaxe:

NomDeFonction(parametres):TypeDuResultat == CorpsDeLaFonction

La définition d'un constructeur de domaine obéit à la même syntaxe:

NomDeConstructeur(parametres): uneCategorie == uneCapsule

```
Polynomes(R:Ring, VarSet: OrderedSet): Public == Private where
   Public  == Join(Algebra(R), Ring)
   Private == add ...  -- implantation des fonctions
```

### 1.7.2 Une syntaxe dépouillée

Dans l'exemple précédent, contrairement à ce qu'on pourrait penser, **Public** et **Private** ne sont pas des mots réservés du langage; n'importe quel identificateur peut être employé. La clause **Where** qui permet de redéfinir les mots **Public** et **Private** n'est pas indispensable ; on l'utilise ici pour améliorer la clarté du programme. On aurait pu écrire:

```
Polynomes(R:Ring, VarSet: OrderedSet):Join(Algebra(R), Ring)
 == add
     ...  -- implantation des fonctions
```

La clause **where** peut en effet servir dans un tout autre contexte tel que:

```
x := a + b where
   a := 2
   b := 3
```

Les programmeurs en SCRATCHPAD utilisent en général la clause **where** pour augmenter la lisibilité des programmes. Ce point de vue est étroit; cette clause serait très puissante dans le cadre de la programmation parallèle: elle exprime que deux calculs peuvent être menés de manière indépendante.

On observe que la *carte syntaxique* de SCRATCHPAD est exagérément simple avec peu de mots réservés [4]. Il s'ensuit que le compilateur de SCRATCHPAD ne peut pas localiser précisément les erreurs figurant dans le fichier source des programmes. Une bonne partie du mauvais fonctionnement du compilateur vient du fait que la syntaxe du langage est trop peu contraignante:

1. Les blocs ne sont pas parenthésés par des mots réservés du genre `IF ... ENDIF`. Le compilateur se base sur l'indentation du texte source.
2. Certaines parenthèses sont superflues;
   l'instruction `y:= f g x` est équivalente à `y := f(g(x))`
3. Les instructions ne sont pas obligatoirement séparées par un caractère spécial (par ex: point-virgule).

### 1.7.3 Des primitives puissantes de manipulation de listes

Le langage SCRATCHPAD est bien adapté à la manipulation des listes et des suites (Voir le constructeur de domaine: **Stream**).

```
(2) ->double x == 2*x
                                                    Type: Void

(3) ->double! [1..5]
   Compiling function double with type Integer -> Integer
```

[4] La palme revient au langage LISP où le seul contrôle syntaxique à effectuer concerne le parenthésage des expressions.

```
   (3)  [2,4,6,8,10]
                                                    Type: List Integer

(4) ->+/ [1..4]

   (4)  10
                                                    Type: Integer

(5) ->[double(i) for i in 1.. | prime?(i)]

   (5)  [4,6,10,14,22,26,34,38,46,58,...]
                                                    Type: Stream PositiveInteger
```

Un **Stream** est une suite d'objets potentiellement infine. Les termes de la suite déjà calculés sont mémorisés dans une liste, les autres sont potentiellement calculables par une formule. Seuls les termes utilisés sont évalués et un même terme n'est jamais évalué deux fois. Cette technique d'évaluation est quelquefois appelée *évaluation paresseuse* ou encore évaluation par *nécessité*.

Il est clair que ces mécanismes font de SCRATCHPAD un langage très puissant et en avance sur son temps; on conçoit qu'il soit assez difficile à compiler de façon fiable.

# 2

# Polynômes et séries formelles

## 2.1 L'algèbre des séries formelles

### 2.1.1 Définitions

Une *série formelle* $S$ à coefficients dans l'anneau $R$ est une application de $X^*$ dans $R$ qui à un mot $w \in X^*$ associe l'élément $S(w)$, noté $< S|w >$, et appelé *coefficient* du mot $w$ dans la série $S$. Cette série sera notée formellement :

$$S = \sum_{w \in X^*} < S|w > w.$$

L'ensemble des séries formelles sera noté $R\langle\langle X \rangle\rangle$. (voir [1, 31]).

Le *support* d'une série $S$ est le langage :

$$\mathrm{Supp(S)} = \{ \mathrm{w} \in \mathrm{X^*} \mid < \mathrm{S|w} > \neq 0 \}.$$

Une série sera dite *propre* si son terme constant est nul ($< S|\varepsilon >= 0$).

Un *polynôme* est une série formelle de support fini. Ce qui revient à dire qu'un polynôme est une série dont tous les coefficients sont nuls sauf un nombre fini. On notera l'ensemble des polynômes $R\langle X \rangle$, c'est une sous-algèbre de $R\langle\langle X \rangle\rangle$.

Le *degré* d'un polynôme $P \in R\langle X \rangle$ est défini par :

$$\deg(P) = \begin{cases} -\infty, & \text{si } P = 0, \\ \sup\{|w| \text{ avec } w \in \mathrm{Supp}(P)\}, & \text{si } P \neq 0. \end{cases}$$

### 2.1.2 Opérations

En ce qui concerne les diverses opérations sur les séries formelles, on pourra consulter l'article de M.Fliess [7].

#### 2.1.2.1 Produit scalaire

C'est une forme bilinéaire symétrique définie sur les mots par:

$$\forall u, v \in X^*, \qquad < u|v > = \delta_u^v$$

Elle se prolonge par linéarité en produit scalaire d'une série $S$ par un polynôme $P$:

$$< S|P > = \sum_{w \in X^*} < S|w >< P|w >$$

### 2.1.2.2 Calcul des résiduels

**Définition 2.1.1** *Soit $u \in X^*$ un mot et $x \in X$ une lettre. On note $u \triangleright x$ (resp. $x \triangleleft u$) le résiduel ou "reste" à droite (resp. à gauche) de $u$ par $x$, défini par :*

$$u \triangleright x = \begin{cases} v & si \quad u = xv, \\ 0 & sinon. \end{cases} \qquad \left( resp.\ x \triangleleft u = \begin{cases} v & si \quad u = vx, \\ 0 & sinon. \end{cases} \right) \tag{2.1}$$

On étend facilement cette notion aux résiduels par les mots en utilisant la propriété suivante : $\forall u \in X^*,\ \forall x, y \in X,$

$$u \triangleright xy = (u \triangleright x) \triangleright y \quad (\text{resp. } xy \triangleleft u = x \triangleleft (y \triangleleft u)).$$

Ainsi, $\forall w \in X^*$,

$$w \triangleright u = \begin{cases} v & \text{si} \quad w = uv, \\ 0 & \text{sinon.} \end{cases} \qquad \left( \text{resp. } u \triangleleft w = \begin{cases} v & \text{si} \quad w = vu, \\ 0 & \text{sinon.} \end{cases} \right) \tag{2.2}$$

Par linéarité, on peut calculer facilement le reste d'une série $S$ à droite (resp. à gauche) par un mot $u$ en posant :

$$S \triangleright u = \sum_{w \in X^*} < S|w > w \triangleright u \qquad \left( \text{resp. } u \triangleleft S = \sum_{w \in X^*} < S|w > u \triangleleft w \right) \tag{2.3}$$

Pour terminer, on peut définir également le reste d'une série $S$ à droite (resp. à gauche) par un polynôme $P$ en posant :

$$S \triangleright P = \sum_{u \in X^*} < P|u > S \triangleright u \qquad \left( \text{resp. } P \triangleleft S = \sum_{u \in X^*} < P|u > u \triangleleft S \right) \tag{2.4}$$

### 2.1.2.3 Lemme de reconstruction

Ayant les résiduels de $S$ à droite par les lettres $z \in X$ et le terme constant $< S|\varepsilon >$ de $S$, on peut reconstruire $S$. On a le lemme suivant [14] dit *lemme de reconstruction.*

**Lemme 2.1.1** *Soit $S \in R\langle\langle X \rangle\rangle$ une série formelle et soit $< S|\varepsilon >$ son terme constant. Alors*

$$S \ = \ < S|\varepsilon > + \sum_{z \in X} z(S \triangleright z).$$

Ce lemme motive l'idée de représenter les polynômes non commutatifs de manière récursive, par leur terme constant et leurs résiduels à droite [27]. On verra par la suite que cette représentation a des avantages énormes.

### 2.1.2.4 Produit de Cauchy

Le *produit de Cauchy* pour les séries formelles est une extension du produit de concaténation défini sur les mots. Soient $S, T \in R\langle\langle X \rangle\rangle$, deux séries formelles non commutatives. Le produit

de Cauchy de $S$ par $T$, noté $S \cdot T$ ou tout simplement $ST$ est défini par :

$$S \cdot T = \sum_{w \in X^*} \left( \sum_{u.v=w} < S|u >< T|v > \right) w.$$

Ce qui peut aussi s'écrire :

$$\forall S, T \in R\langle\langle X \rangle\rangle, \ \forall w \in X^*, \quad < S \cdot T|w >= \sum_{u \cdot v = w} < S|u >< T|v > .$$

On vérifie que la somme ci-dessus ne porte que sur un nombre fini de termes. Ce produit est évidemment associatif mais non commutatif. L'ensemble des séries formelles muni de ce produit est une *algèbre associative.*

On a, pour des séries $S$ et $T$ , une lettre $z$ et des polynômes $P$ et $Q$, les propriétés suivantes:

$$\begin{aligned} (S \cdot T) \triangleright z &= (S \triangleright z) \cdot T + < S|\varepsilon > (T \triangleright z) \\ (S \triangleright P) \triangleright Q &= S \triangleright (PQ) \\ (P \triangleleft S) \triangleright Q &= P \triangleleft (S \triangleright Q) \\ < S|PQ > &= < Q \triangleleft S|P > = < S \triangleright P|Q > \end{aligned}$$

### 2.1.2.5 Produit de mélange

Le *produit de mélange*[1] (M.Fliess [8]; G.Jacob & N.Oussous [14]) est défini récursivement, pour les mots, par :

$$\begin{cases} \forall w \in X^*, & w \sqcup \varepsilon = \varepsilon \sqcup w = w, \quad (\varepsilon \text{ étant le mot vide}). \\ \forall u, v \in X^*, \ \forall x, y \in X, & (xu) \sqcup (yv) = x(u \sqcup (yv)) + y((xu) \sqcup v). \end{cases} \tag{2.5}$$

Cette définition du produit de mélange de deux mots nous donne directement un algorithme récursif simple mais coûteux en temps de calcul et en place mémoire.

Ce produit est commutatif [2] et associatif ; on peut l'étendre facilement aux polynômes et aux séries en posant, pour $S, T \in R\langle\langle X \rangle\rangle$ des séries :

$$S \sqcup T = \sum_{u,v \in X^*} < S|u >< T|v > u \sqcup v.$$

**Remarque :**

Lorsque l'une des séries $S$ ou $T$ est une constante, le produit de mélange devient le produit ordinaire d'une série par une constante.

Le produit de mélange [14] peut être redéfini en posant, pour des mots $u$ et $v$ et une lettre quelconque $z$ :

$$\begin{aligned} (u \sqcup v) \triangleright z &= (u \triangleright z) \sqcup v + u \sqcup (v \triangleright z), \\ < u \sqcup v|\varepsilon > &= \begin{cases} 0 & \text{si} \quad uv \neq \varepsilon, \\ 1 & \text{si} \quad uv = \varepsilon. \end{cases} \end{aligned} \tag{2.6}$$

[1] En anglais : Shuffle product.

[2] lorsque l'anneau $R$ est commutatif.

Ainsi, cette action est une dérivation pour le produit de mélange. On peut donc la noter : $\partial_z \equiv \triangleright z$.
Cette propriété démontrée sur les mots, se prolonge par linéarité sur les séries.

**Lemme 2.1.2** *Si $z \in X$, alors $\partial_z$ est une dérivation dans $R\langle\langle X\rangle\rangle$ pour le produit de mélange ie.*

$$(S \sqcup T) \triangleright z = (S \triangleright z) \sqcup T + S \sqcup (T \triangleright z)$$

Ce lemme est bien sûr valable pour les polynômes.

## 2.2 Implantation en Scratchpad

### 2.2.1 L'interface: les catégories à prévoir

#### 2.2.1.1 Importance de l'interface

Un des buts de la programmation orientée objet est la réutilisabilité des composants logiciels. L'expérience prouve que pour des projets de grande envergure (un projet comme MACSYMA a demandé 100 homme.année), la difficulté principale est de maintenir une grande cohérence entre les différents modules. Il arrive fréquemment que la taille d'un projet augmentant, les concepteurs n'osent plus modifier un quelconque module de peur de ne pouvoir maîtriser des conséquences imprévues quant au comportement de l'ensemble du système; un tel projet est alors à l'article de la mort.

Bref le premier souci du concepteur ne doit pas être de nature algorithmique (il est presque toujours possible de modifier après-coup tel algorithme ou telle structure de données qui se trouve être critique) mais **d'identifier clairement les domaines à implanter et de fixer leur interface** ie. fixer le nom et la signature des fonctions qu'ils exportent.

#### 2.2.1.2 Une interface commune

Il se trouve que l'étude des systèmes dynamiques non linéaires requiert l'utilisation d' outils tels que **polynômes**, **séries formelles rationnelles** et **R-automates** et que les primitives de base de manipulation de ces objets sont presque les mêmes.

La définition d'une catégorie en SCRATCHPAD commune pour ces trois domaines est un gage de cohérence. Elle contient la spécification des opérations de base ayant un sens pour chacun des domaines précités.

```
)abbrev category XFALG XFreeAlgebra
XFreeAlgebra(vl:OrderedSet, R:Ring):Category == Catdef where
   OFMON1 ==> OrderedFreeMonoid1        ++ monoide ordonne libre
   NNI    ==> NonNegativeInteger

   Catdef == Join(Ring,Algebra(R))
     with
       "*": (vl,$) -> $
```

```
    "*": ($, R) -> $                ++ utile si R est non commutatif
    mindeg: $ -> OFMON1(vl)         ++ Attention pour le polynome nul
    mindegMonomial: $ -> Record(k: OFMON1(vl), c: R)
    coef  : ($,OFMON1(vl)) -> R     ++ coefficient d'un mot
    coef  : ($,$) -> R
    lquo  : ($,vl) -> $             ++ residuel a gauche d'une lettre
    lquo  : ($,OFMON1(vl)) -> $     ++ residuel a gauche d'un mot
    lquo  : ($,$) -> $
    rquo  : ($,vl) -> $
    rquo  : ($,OFMON1(vl)) -> $
    rquo  : ($,$) -> $
    monom : (OFMON1(vl), R) -> $
    mirror: $ -> $                  ++ miroir sur chaque mot
    coerce : vl -> $
    coerce : OFMON1(vl) -> $
    constant?:$ -> Boolean          ++ renvoie vrai pour un element de R
    constant: $ -> R                ++ renvoie le terme constant
    quasiRegular? : $ -> Boolean    ++ vrai si le terme constant est nul
    quasiRegular : $ -> $           ++ supprime le terme constant
    sh :($,$) -> $                  ++ produit de melange
    sh :($,NNI) -> $
    map   : (R -> R, $) -> $        ++ transf. des coef par une fonction
    varList: $ -> List vl           ++ liste des variables
    if R has Field then "/" : ($,R) -> $
  -- Attributs
    if R has noZeroDivs then noZeroDivs
```

#### 2.2.1.3 La catégorie des polynômes

Cette catégorie reprend les spécifications précédentes en y ajoutant ce qui est propre aux polynômes (calcul du mot de plus haut degré ...).

```
)abbrev category XPOLYC XPolynomialsCat

XPolynomialsCat(vl:OrderedSet,R:Ring):Category == XFreeAlgebra(vl,R) with
    maxdeg: $ -> OrderedFreeMonoid1 vl
    degree: $ -> NonNegativeInteger
    trunc : ($ , NonNegativeInteger) -> $
```

### 2.2.2 L'implantation des polynômes en représentation distribuée

#### 2.2.2.1 Tout commence avec des mots

Dans cette représentation, un polynôme est considéré comme une combinaison linéaire finie de mots. Il convient donc d'implanter en premier la gestion des mots construits sur un alphabet donné; celle-ci est réalisée par le constructeur `OFMON1`:

```
)abbrev domain OFMON1 OrderedFreeMonoid1
OrderedFreeMonoid1(S: OrderedSet): OFMcategory == OFMdefinition where
    NNI ==> NonNegativeInteger

    OFMcategory == OrderedMonoid with
        coerce: S -> $
        "*":    (S, $) -> $
        "**":   (S, NNI) -> $
        first: $ -> S
        rest:  $ -> $
        mirror: $ -> $
        lexico: ($,$) -> Boolean

        -- highest common left/right factor
        hclf:   ($, $) -> $
        hcrf:   ($, $) -> $

        -- q = lquo(a*q, a)  q = rquo(q*a, a)
        lquo:   ($, $) -> Union($, "failed")
        rquo:   ($, $) -> Union($, "failed")
        lquo:   ($, S) -> Union($, "failed")
        rquo:   ($, S) -> Union($, "failed")

        -- (l, r) := (l*a*r) div a
        "div":   ($, $) -> Union(Record(lm: $, rm: $), "failed")
        overlap: ($, $) -> Record(lm: $, mm: $, rm: $)
        size:   $ -> NNI
        nthExpon:  ($, Integer) -> NNI
        nthFactor: ($, Integer) -> S
        listOfFactors: $ -> List Record(gen: S, exp: NNI)
        length: $ -> NNI

        -- liste des variables d'un mot
        varList: $ -> List S

    OFMdefinition == FreeMonoid(S) add
      -- representation
        Term := Record(gen: S, exp:Integer )
        Rep:= List Term

      -- definitions
        length x ==
          n: Integer := +/[f.exp for f in x]
          n::NNI
      ....'
```

Cette implantation est la reprise de celle fournie en "Standard" en SCRATCHPAD. Elle est assez complète et permet, en particulier de programmer facilement la détection des paires critiques intervenant dans le calcul d'une base standard voir(chapitre 4 page 59) d'un idéal

bilatère [3].

Le choix de représenter les mots par une liste de couples **(lettre, exposant)** est judicieux pour traiter de longues répétitions de lettres ; ce cas se produit assez peu souvent dans la pratique. Il a, par contre, le grave inconvénient de ralentir la procédure de comparaison de deux mots. Or l'efficacité de cette procédure est fondamentale pour tous les calculs en représentation distribuée (voir 2.2.2.3).

#### 2.2.2.2 L'algèbre des polynômes construits sur un monoïde quelconque

Tout polynôme $p \in R\langle X \rangle$ est une combinaison linéaire des mots de $X^*$ et peut donc être représenté par une liste de termes contenant chacun un mot et son coefficient respectif.

Si

$$p = \sum_{i=1}^{n} a_i w_i$$

alors p est représenté par la liste:

$$((a_0, w_0), (a_1, w_1), \ \ldots\ , (a_n, w_n))$$

Le fait d'utiliser un **monoïde libre** n'est important que pour quelques fonctions (résiduels, shuffle). Toutes les autres peuvent être implantées sur un **monoïde quelconque**; c'est la raison d'être du constructeur XPR:

```
)abbrev domain XPR
XPolynomialRing(R:Ring,E:OrderedMonoid): T == C where
  T == Join(Ring,Algebra(R)) with
    --operations
      "*": ($,R) -> $
      "#": $ -> NonNegativeInteger
      coerce: E -> $
      maxdeg: $ -> E
      mindeg: $ -> E
      ...
    --assertions
      if R has noZeroDivs then noZeroDivs
      if R has unitsKnown then unitsKnown
      if R has canonicalUnitNormal then canonicalUnitNormal

  C == FreeModule1(R,E) add
    --representations
       Term:=  Record(k:E,c:R)
       Rep:=  List Term
    --define
       1  == [[1$E,1$R]]
        ...
```

[3] en variables non commutatives.

Cette unité de compilation est la copie d'une autre existant pour l'algèbre commutative; malheureusement, l'héritage est impossible compte–tenu des remarques faites en 1.5.4.

#### 2.2.2.3 Importance de l'ordre sur le monoïde:

L'existence d'un ordre sur les mots conditionne l'efficacité de l'algorithme d'addition de deux polynômes. Nous utilisons un algorithme très classique de fusion de deux listes triées; il faut veiller à éliminer de la somme, les monômes ayant un coefficient nul.

Voici la solution retenue dans l'unité `FreeModule(R: RING,S: ORDSET)`

```
x + y  ==
   null x => y
   null y => x
   y.first.k > x.first.k => [y.first,:(x + y.rest)]
   x.first.k > y.first.k => [x.first,:(x.rest + y)]
   r:= x.first.c + y.first.c
   r = 0 => x.rest + y.rest
   [[x.first.k,r],:(x.rest + y.rest)]
```

On remarquera l'utilisation de la forme `[a,:b]` qui a le même effet que l'expression `(cons a b)` dans le langage Lisp.

Nous constatons que la primitive la plus critique (temps d'exécution) est la comparaison des mots qui renvoie bien sûr à la comparaison des lettres d'un alphabet. C'est ici qu'il conviendrait *d'optimiser les algorithmes.*

Une autre question concerne le problème de la place allouée en mémoire dynamique lors d'une addition de deux polynômes. Le cas le plus défavorable se produit lorsqu'on additionne un polynôme $p$ de degré élevé avec un polynôme $q$ de degré faible. Toute la représentation interne du polynôme $p$ est dupliquée en mémoire.

#### 2.2.2.4 L'algèbre des polynômes construits sur un monoïde libre

Cette implantation hérite de l'implantation des polynômes construits sur un monoïde quelconque ; il suffit d'y ajouter quelques fonctionnalités spécifiques : calcul des résiduels et du produit de mélange. Voici une partie du texte source :

```
)abbrev domain XDPOLY XDistribuedPolynomial
XDistribuedPolynomial(vl:OrderedSet,R:Ring): XDPcat == XDPdef where

  WORD ==> OrderedFreeMonoid1(vl)
  NNI  ==> NonNegativeInteger

  XDPcat == Join(FreeModuleCat(R, WORD), XPolynomialsCat(vl,R)) with
    ListOfTerms : $ -> List Record(k:WORD, c:R)
    numberOfMonomials: $ -> NNI
    mindegMonomial: $ -> Record(k:WORD, c:R)
    maxdegMonomial: $ -> Record(k:WORD, c:R)
```

```
XDPdef == XPolynomialRing(R,WORD) add
     import( WORD)
  -- Representation
     Term :=  Record(k:WORD, c:R)
     Rep  :=  List Term
  ....
```

Certaines primitives ont été implantées pour permettre un calcul facile des bases-standard d'espaces vectoriels de polynômes (voir page 69).

### 2.2.3 L'implantation des polynômes en représentation récursive

#### 2.2.3.1 Les types récursifs en Scratchpad

SCRATCHPAD permet de définir facilement des types fondamentalement récursifs et cela, sans utiliser la notion de pointeur.

Voici comment les concepteurs du système SCRATCHPAD définissent la représentation interne des polynômes commutatifs :

```
SparseMultivariatePolynomial(R: Ring,VarSet: OrderedSet): C == T where
  C == MPolyCat(VarSet,R)
  T == add
    --representations
      D := SparseUnivariatePolynomial($)
      VPoly:=  Record(v:VarSet,ts:D)
      Rep:=  Union(R,VPoly)
    --definitions
      .....
```

L'idée qui est exprimée est la suivante :

> On fixe un ordre sur les variables ; la plus grande des variables figurant dans un polynôme est appelée sa *variable principale.*
>
> Un polynôme est de l'un des deux types suivants :
>
> - `R` s'il est constant (il appartient donc à l'anneau de base `R`),
> - `VPoly` s'il comporte au moins une variable.
>
> Un polynôme à plusieurs variables est considéré comme *un polynôme à une variable* (la variable principale) dont les coefficients sont des polynômes construits avec les autres variables.
>
> Se donner un polynôme de type `VPoly` revient à se donner sa variable principale et un polynôme à une variable représenté par la liste de ses coefficients.

Cette possibilité de définir facilement des domaines récursifs est souvent utilisée. Ainsi :

*Une extension algébrique de degré fini d'un corps k est*

- *soit le corps k lui-même,*
- *soit une extension simple d'une autre extension algébrique de degré fini.*

#### 2.2.3.2 La représentation récursive

Nous pouvons partir du lemme de reconstruction d'un polynôme $p \in R\langle X\rangle$ :

$$p = p_0 + \sum_{x \in X} x(p \triangleright x)$$

| | |
|---|---|
| $p_0 =< p\|\varepsilon >$ | représente le terme constant, |
| $p \triangleright x$ | représente le résiduel à droite de $p$ pour la lettre $x$, |
| $\sum_{x \in X} x(p \triangleright x)$ | constitue la partie propre de $p$. |

Celle-ci est une "combinaison linéaire" des lettres $x$ de l'alphabet $X$, les coefficients étant eux-mêmes des polynômes; il se trouve qu'il existe déjà dans SCRATCHPAD un constructeur de domaine `FreeModule` permettant de gérer ces combinaisons linéaires.

**2.2.3.2.1 Exemple** Soit

$$\begin{aligned} p &= 5 + a^2 + 2ab + 3b \\ &= 5 + a(a + 2b) + b(3) \end{aligned}$$

On a alors:

$$\begin{aligned} p \triangleright a &= a + 2b \\ p \triangleright b &= 3 \end{aligned}$$

On peut alors représenter $p$ par la liste:

$$(5, (a, p \triangleright a), (b, p \triangleright b))$$

Il suffit alors d'itérer la méthode pour représenter les résiduels de $p$ :

$p \triangleright a$ est représenté par $(0, ((a, 1), (b, 2)))$,
$p \triangleright b$ est représenté par 3.

**2.2.3.2.2 Formellement** Un polynôme de $R\langle X\rangle$ est

- soit un élément de l'anneau $R$,
- soit un *vrai polynôme.*

Un *vrai polynôme* comporte :

- un terme constant (qui peut être nul),
- une partie propre qui est un *module libre* construit sur $X$, les coefficients étant des polynômes.

#### 2.2.3.2.3 Tout ceci se traduit très naturellement en SCRATCHPAD [4]

```
)abbrev domain XRPOLY     XRecursivePolynomial

XRecursivePolynomial(VarSet:OrderedSet,R:Ring):  Xcat == Xdef where
  XDPOLY ==> XDistribuedPolynomial(VarSet, R)
  TERM   ==> Record(k:VarSet , c:$)
  LTERMS ==> List(TERM)
  REGPOLY==> FreeModule1($, VarSet)
  VPOLY  ==> Record(c0:R, reg:REGPOLY)

  Xcat == XPolynomialsCat(VarSet,R) with
       extend: $ -> XDPOLY
       unextend : XDPOLY -> $
       RemainderList: $ -> LTERMS

  Xdef == add
       import(VPOLY)

    -- representation
       Rep      := Union(R,VPOLY)
    ....
```

Tout ceci est de la bonne programmation *orientée objet* puisque nous réutilisons pour les calculs *linéaires*, les fonctions implantées dans `FreeModule1`.

### 2.2.3.3 SCRATCHPAD vous rend la vie simple

Le constructeur `XRPOLY` demande qu'on lui passe en paramètre l'alphabet `VarSet` fixant l'ensemble des variables utilisables, ce qui n'est pas toujours très pratique, surtout si l'on désire employer des noms de variable quelconques. La solution consiste à définir un nouveau constructeur `XP` qui n'est pas paramétré par l'alphabet.

La définition en est très simple; elle se fait par héritage de `XRPOLY` en utilisant le domaine `SortedExpression` appartenant à la catégorie des `OrderedSet` [5].

```
)abbrev domain XP XPolynomial
XPolynomial(R:Ring) == XRecursivePolynomial(SortedExpressions(),R)
```

## 2.2.4 Schéma récapitulatif

Nous donnons, dans la figure 2.1 de la page 36 une architecture de l'implantation des polynômes non commutatifs.

---

[4] L'alphabet $X$ est noté `VarSet`

[5] les `SortedExpression` sont des expressions quelconques triées par ordre alphabétique.

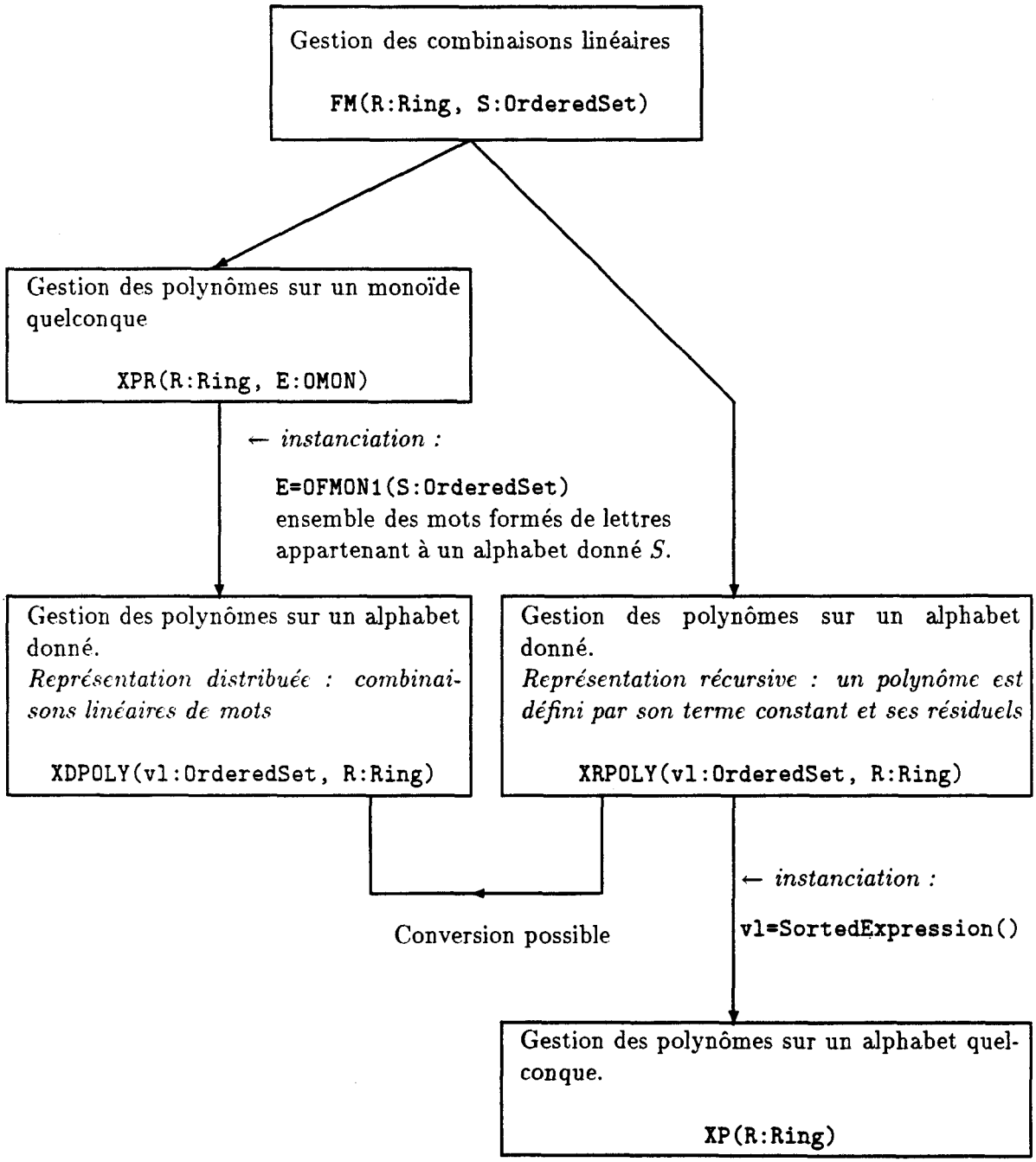



Figure 2.1 : Architecture de l'implantation des polynômes non commutatifs

### 2.2.5 Quelques algorithmes

#### 2.2.5.1 Idée de base en représentation récursive

Le calcul d'une fonction $f(p)$ se fait par appel récursif de cette même fonction $f$ sur les résiduels de $p$.
La récursivité s'arrête lorsque le polynôme $p$ est constant.

Soit à calculer le plus grand mot (ordre lexicographique par longueur) appartenant au support d'un polynôme donné $p$ ; voici une solution possible :

```
maxdeg p ==
   p case R => 1$Word
   "max"/[(t.k) *$Word maxdeg(t.c) for t in RemainderList p]
```

#### 2.2.5.2 Calcul du shuffle en représentation récursive

Le shuffle $p \sqcup q$ est défini par :

- son terme constant obtenu comme un produit dans l'anneau des coefficients:

$$< (p \sqcup q)|\varepsilon > = < p|\varepsilon > \cdot < q|\varepsilon >$$

- son résiduel pour chaque lettre $x$ de l'alphabet :

$$(p \sqcup q) \triangleright x = (p \triangleright x) \sqcup q + p \sqcup (q \triangleright x)$$

*Les résiduels $p \triangleright x$ et $q \triangleright x$ ne nécessitent aucun calcul puisqu'ils figurent explicitement dans les représentations récursives de $p$ et $q$.*

Les appels récursifs du calcul du shuffle de deux polynômes s'arrêtent lorsque l'un des polynômes est constant. Si $p$ (ou $q$) est constant, $p \sqcup q = p \cdot q$.

```
sh(p1,p2) ==
  p1 case R => p1::R * p2
  p2 case R => p1 * p2::R
  lt1 := ListOfTerms p1.reg ; lt2 := ListOfTerms p2.reg
  x := makePoly [[t.k, sh(t.c,p2)]$Term for t in lt1]
  y := makePoly [[t.k, sh(t.c,p1)]$Term for t in lt2]
  [p1.c0*p2.c0, x + y]$VPoly
```

La comparaison avec l'algorithme en représentation distribuée est aisée ; le shuffle de deux polynômes s'obtient à partir du shuffle sur les mots. Soient

$$p = \sum_{i \in I} a_i u_i \quad \text{et} \quad q = \sum_{j \in J} b_j v_j \qquad (I, J \quad \text{fini})$$

On a alors:

$$p \sqcup q = \sum_{i \in I, j \in J} a_i b_j (u_i \sqcup v_j).$$

#### 2.2.5.3 Calcul du produit de Cauchy de deux polynômes

L'idée de base est la même :

$$(p \cdot q) \triangleright x = (p \triangleright x) \cdot q + p_0(q \triangleright x)$$

Voici à titre d'exemple une solution possible:

```
p1:$ * p2:$ ==
  p1 case R => p1::R * p2
  p2 case R => p1 * p2::R
  x:REGPOLY := p1.reg *$REGPOLY p2
  y:REGPOLY := (p1.c0) *$REGPOLY p2.reg
  [ p1.c0 * p2.c0 , x+y ]$VPOLY
```

#### 2.2.5.4 Calcul du résiduel à gauche d'un polynôme

La représentation par les résiduels à droite rend malaisé le calcul du résiduel à gauche d'un polynôme $p$ par un polynôme $q$ ; ce calcul est néanmoins possible en partant des formules:

$$\begin{aligned} \forall x \in X, \quad (q \triangleleft p) \triangleright x &= q \triangleleft (p \triangleright x) \\ < (q \triangleleft p) | \varepsilon > &= < (p \triangleright q) | \varepsilon > = < p | q > \end{aligned}$$

### 2.2.6 Test et comparaison

Nous donnons dans l'annexe B, page 125 un petit fichier de commandes permettant de calculer le shuffle de deux polynômes, l'un de degré 6 et l'autre de degré 2. Tous les calculs sont faits en double, c'est-à-dire avec les deux représentations, ceci afin de comparer les résultats obtenus et le temps consommé.

> **On constate pour un même calcul de shuffle, que le temps nécessaire peut varier dans un facteur 30 selon la représentation choisie.**

Le tableau 2.1 contient des statistiques sur les temps de calcul pour les deux représentations (récursive et distribuée).

Dans ce tableau, nous avons choisi de donner les temps de calcul du produit de mélange car c'est l'opération qui coûte le plus cher dans le traitement des polynômes non commutatifs. Ce temps dépend essentiellement du nombre et de la longueur des mots figurant dans les polynômes à "mélanger". De toute façon, si l'on exécute deux fois le même calcul, les temps de réponse peuvent être sensiblement différents car des facteurs assez aléatoires interviennent: défauts de page, garbage collector ...

| Degrés | Nombres de mots | | Repr. dist. | Repr. réc. |
|---|---|---|---|---|
| | poly. $P_1$ | poly. $P_2$ | temps en sec. | temps en sec. |
| 1 | 3 | 2 | 0 | 0 |
| 2 | 7 | 6 | 0.66 | 0 |
| 3 | 14 | 17 | 40 | 1 |
| 4 | 28 | 39 | 1377 | 40 |

Tableau 2.1 : Tableau comparatif

### 2.2.7 Evaluation, codage des intégrales itérées

#### 2.2.7.1 Le problème général de l'évaluation

Toute application de $X$ dans une $R$-algèbre quelconque $\mathcal{A}$ se prolonge de façon unique en un morphisme d'algèbre, de $R\langle X\rangle$ dans $\mathcal{A}$. Autrement dit, l'algèbre $R\langle X\rangle$ est une **algèbre initiale** dans la catégorie des $R$-algèbres associatives.

Intuitivement cela signifie que si l'on sait évaluer chaque lettre de l'alphabet $X$, on sait évaluer chaque polynôme de $R\langle X\rangle$.

SCRATCHPAD permet d'exprimer cette idée très générale, ce qui évite de multiplier les fonctions d'évaluation lorsque l'on désire interpréter les lettres d'un polynôme comme

1. des constantes de l'anneau des coefficients,
2. des matrices sur l'anneau des coefficients,
3. des polynômes sur l'anneau des coefficients,
4. des opérateurs différentiels,
5. des opérateurs d'intégration.

Voici un paquetage qui implante l'algorithme d'évaluation au niveau le plus général.

```
)abbrev package HORNER HornerEvaluationPackage

HornerEvaluationPackage(VarSet,R,A): Public == Private where
   VarSet : OrderedSet
   R      : Ring
   A      : Algebra(R)
   SUP    ==> SparseUnivariatePolynomial(R)
   SMP    ==> SparseMultivariatePolynomial(R,VarSet)
   XRPOLY ==> XRecursivePolynomial(VarSet,R)
   XDPOLY ==> XDistribuedPolynomial(VarSet,R)
   NNI    ==> NonNegativeInteger

   Public ==> with
     eval: (SUP, A) -> A      -- substitution dans un polynome a une variable
     eval: (SMP, VarSet -> A) -> A  -- substitution de plusieurs variables
```

```
    eval: (XRPOLY, VarSet -> A) -> A
    eval: (XDPOLY, VarSet -> A) -> A

  Private ==> add
    eval(p: XDPOLY, i:VarSet -> A):A ==
    ....
```

Il est à noter que l'on passe en paramètre la fonction d'interprétation `i:VarSet -> A`, ce qui permet de travailler sur un alphabet infini [6].

#### 2.2.7.2 Application: codage des intégrales itérées

Afin de simplifier l'explication, nous allons nous restreindre à un alphabet de deux lettres $X = \{z_1, z_2\}$ et à une $\mathbb{R}$-algèbre $\mathcal{A}$ engendrée par deux opérateurs d'intégration $J_1$ et $J_2$.

Les opérateurs d'intégration $J_1$ et $J_2$ sont définis comme suit:

$$\forall f : \mathbb{R} \to \mathbb{R}, \quad J_1(f) = \int_0^t a_1(\tau) f(\tau) d\tau$$

$$\forall f : \mathbb{R} \to \mathbb{R}, \quad J_2(f) = \int_0^t a_2(\tau) f(\tau) d\tau$$

($a_1$ et $a_2$ désignent deux fonctions données de $\mathbb{R}$ dans $\mathbb{R}$)

Choisissons de coder ces deux opérateurs par les lettres $z_1$ et $z_2$:

$$\begin{cases} z_1 \longrightarrow J_1 \\ z_2 \longrightarrow J_2 \end{cases}$$

On pourra alors considérer tout polynôme de $\mathbb{R}\langle X \rangle$ comme le codage d'un opérateur d'intégration obtenu par **compositions** ou **sommes** des opérateurs $J_1$ et $J_2$.

A titre d'exemple, le polynôme $p = 2z_1z_2 + 5$ code l'opérateur:

$$\begin{aligned} J_p &= 2J_1 \circ J_2 + 5 \\ J_p(f) &= 2\int_0^t a_1(\tau) \left( \int_0^\tau a_2(\sigma) f(\sigma) d\sigma \right) d\tau + 5f. \end{aligned}$$

## 2.3 Conclusion et perspectives

En ce qui nous concerne, la tâche en algèbre non commutative est immense. Ici, nous avons implanté les polynômes non commutatifs et nous avons montré le rôle joué par la représentation interne dans l'efficacité des algorithmes.

Les calculs du produit de mélange sont particulièrement coûteux en temps et en place mémoire. Nous verrons dans les sections 5.3 et 6.5.1 que l'on peut représenter les polynômes de $R\langle X \rangle$ comme des polynômes commutatifs définis sur un alphabet infini (une base de l'algèbre de mélange). Le calcul du "shuffle" se ramène a un simple produit en variables commutatives ; malheureusement dans ce cas, le produit de Cauchy devient coûteux en temps de calcul.

[6] l'alphabet est un *domaine* et non une *liste de variables*

# 3

# Algèbres et polynômes de Lie

## 3.1 Motivation

Le but de ce chapitre est de fournir les principaux théorèmes permettant d'implanter efficacement les polynômes de l'algèbre associative $K\langle X\rangle$ dans une base de Poincaré–Birkoff–Witt [2, 15]. Nous montrons comment effectuer la conversion des polynômes exprimés dans leur base canonique $X^*$.

Un tel travail n'est possible que si l'on dispose au préalable d'une bonne implantation des polynômes de Lie. Nous avons choisi la base de Lyndon de préférence à la base de Hall, parce qu'elle permet d'effectuer plus facilement certaines conversions. Ce travail prolonge l'article de P.V. Koseleff [18] en donnant un algorithme efficace de calcul du crochet de Lie de deux polynômes exprimés dans la base de Lyndon.

## 3.2 La catégorie des algèbres de Lie

### 3.2.1 Notion d'algèbre de Lie

Soit $\mathcal{L}$ un module sur un anneau unitaire $K$ de caractéristique nulle, muni de l'opération:

$$\mathcal{L} \times \mathcal{L} \longrightarrow \mathcal{L}$$

$$(x, y) \longrightarrow [x, y]$$

On aura quelquefois besoin de diviser les éléments de $K$ par des entiers, ce qui revient à considérer $K$ comme un $\mathbb{Q}$–module.

**Definition 3.2.1** *$\mathcal{L}$ est une* algebre de Lie **ssi**

1. *Le crochet $[x, y]$ est bilinéaire,*
2. $\forall x \in \mathcal{L}, \quad [x, x] = 0,$
3. $\forall x, y, z \in \mathcal{L}, \quad [x,[y,z]] + [y,[z,x]] + [z,[x,y]] = 0,$ *(identité de JACOBI).*

De (1) et (2) il résulte que:

$$\forall x, y \in \mathcal{L}, \quad [x, y] = -[y, x].$$

Il est utile de définir une catégorie commune fixant une interface commune pour les différentes implantations possibles des polynômes de Lie.

```
)abbrev category LALG LieAlgebra
LieAlgebra(R:Ring): Category ==  Module(R) with
  --exports
    construct:  ($,$) -> $   -- crochet de Lie
    if R has Field then
       "/"   :  ($,R) -> $

  --attributs
    NullSquare              -- [x,x] = 0
    JacobiIdentity          -- [x,[y,z]]+[y,[z,x]]+[z,[x,y]] = 0
```

### 3.2.2 Algèbre de Lie libre

Soit $X$ un alphabet totalement ordonné. On désignera par $\mathcal{M}agma\langle X\rangle$ le magma libre des mots complètement parenthésés et par $\mathcal{L}ib_K\langle X\rangle$ l'algèbre construite sur ce magma. L'algèbre de Lie libre, notée $\mathcal{L}ie_K\langle X\rangle$ ou plus simplement $\mathcal{L}ie\langle X\rangle$ est définie comme le quotient:

$$\mathcal{L}ie\langle X\rangle = \mathcal{L}ib_K\langle X\rangle/\mathcal{J}$$

où $\mathcal{J}$ est l'idéal engendré par les éléments de la forme:

$$\begin{array}{lcll} Q(x) & = & [x,x] & \text{pour } x \in \mathcal{L}ib_K\langle X\rangle, \\ J(x,y,z) & = & [x[y,z]]+[y[z,x]]+[z[x,y]] & \text{pour } x,y,z \in \mathcal{L}ib_K\langle X\rangle. \end{array}$$

Tout polynôme de Lie peut être interprété comme un un polynôme en variables non commutatives en considérant les crochets comme des commutateurs:

$$\forall p,q \in \mathcal{L}ie\langle X\rangle, \qquad [p,q] \;=\; pq - qp.$$

Ceci se traduit en SCRATCHPAD par la définition :

```
)abbrev category FLALG  FreeLieAlgebra

FreeLieAlgebra(VarSet:OrderedSet, R:Ring) :Category == CatDef where
   XRPOLY  ==> XRecursivePolynomial(VarSet,R)
   XDPOLY  ==> XDistribuedPolynomial(VarSet,R)

   CatDef ==  Join(LieAlgebra(R), RetractableTo(VarSet)) with
     coef      : (XRPOLY , $) -> R            -- produit scalaire
     coerce    : $ -> XDPOLY
     coerce    : $ -> XRPOLY
     degree    : $ -> NonNegativeInteger
     if R has Module(RationalNumber) then
       Hausdorff : ($,$,PositiveInteger) -> $  -- prod. de Hausdorff tronque
     lquo      : (XRPOLY , $) -> XRPOLY        -- residuel a droite
     LiePoly   : LyndonWord(VarSet) -> $
     mirror    : $ -> $                        -- image miroir
```

```
rquo      : (XRPOLY , $) -> XRPOLY          -- residuel a gauche
trunc     : ($, NonNegativeInteger) -> $    -- troncature
varList   : $ -> List VarSet
```

## 3.3 Bases de l'algèbre de Lie libre

Les deux bases les plus connues sont la base de Lyndon et la base de Hall. Pour des raisons qui seront précisées dans ce chapitre, nous utiliserons la base de Lyndon. En ce qui concerne cette section, on pourra consulter l'ouvrage de M.Lothaire [20], les articles de G. Viennot [34] ou la thèse de N.E. Oussous [24] chapitre IV.

### 3.3.1 Mots de Lyndon

**Definition 3.3.1** *Deux mots $u$ et $v$ sont dits conjugués* **ssi**

$$\exists x, y \in X^* \quad \textit{tels que} \quad u = xy \textit{ et } v = yx.$$

Cette relation est une relation d'équivalence sur $X^*$.

**Exemple:** Les mots $a^2b$ , $aba$ et $ba^2$ sont deux à deux conjugués et forment une classe de conjugaison. Intuitivement, celà consiste à considérer un même mot circulaire.

**Definition 3.3.2** *Un mot $w \in X^*$ est un mot de Lyndon* **ssi** *il vérifie l'une des deux propriétés équivalentes:*

1. *Il est strictement plus petit que tous ses conjugués propres.*
2. *Il est strictement plus petit que tous ses facteurs droits propres.*

Voici à titre d'exemple les mots de Lyndon sur l'alphabet $X = \{a, b\}$, calculés jusqu'à la longueur 5, et triés d'après l'ordre lexicographique par longueur:

$$\{a, b, ab, a^2b, ab^2, a^3b, a^2b^2, ab^3, a^4b, a^3b^2, a^2bab, a^2b^3, abab^2, ab^4\}$$

**Proposition 3.3.1** *Si $u$ et $v$ sont deux mots de Lyndon, alors $uv$ est un mot de Lyndon* **ssi** $u < v$.

### 3.3.2 Crochets de Lyndon

Soit $\delta : \mathcal{M}agma\langle X\rangle \longrightarrow X^*$ l'opération de déparenthésage. L'ensemble des crochets est muni d'un ordre partiel hérité de l'ordre lexicographique sur les mots du monoïde libre ie.

$$\forall u, v \in \mathcal{M}agma\langle X\rangle, \qquad u < v \quad \textbf{ssi} \quad \delta(u) < \delta(v)$$

**Definition 3.3.3** *L'ensemble des crochets de Lyndon est défini récursivement de la façon suivante:*

1. *Les lettres de $X$ sont des crochets de Lyndon.*
2. *Si $a$ est une lettre et si $b$ est un crochet de Lyndon, alors $[a,b]$ est un crochet de Lyndon* **ssi** $a < b$.
3. *Si $[a,b]$ et $c$ sont des crochets de Lyndon, alors $[[a,b],c]$ est un crochet de Lyndon* **ssi** $ab < c \leq b$.

On notera $Lyndon\langle X\rangle$ l'ensemble des crochets de Lyndon définis sur l'alphabet $X$.

Voici les premiers crochets de Lyndon sur $X = \{a,b\}$:

$$\{a, b, [a,b], [a,[a,b]], [[a,b],b], [a,[a,[a,b]]], [a,[[a,b],b]], \ldots\}$$

### 3.3.3 Parenthésage et déparenthésage

#### 3.3.3.1 Déparenthésage des crochets de Lyndon

Par définition, l'opération de déparenthésage $\delta$ transforme tout crochet de $\mathcal{M}agma\langle X\rangle$ en un mot de $X^*$.

**Théorème 3.3.1** *L'opération de déparenthésage définie sur $\mathcal{M}agma\langle X\rangle$ et à valeurs dans $X^*$ réalise une bijection des crochets de Lyndon sur les mots de Lyndon.*

#### 3.3.3.2 Parenthésage des mots de Lyndon

Soit $w$ un mot de Lyndon ; la factorisation $w = lm$ est dite *standard* **ssi** $m$ est le plus long facteur droit propre de $w$ qui est un mot de Lyndon.

L'opération $\sigma$ de parenthésage des mots de Lyndon est définie récursivement comme suit:

$$\begin{array}{rll} \sigma(x) & = x & \text{si } x \text{ est une lettre,} \\ \sigma(lm) & = [\sigma(l), \sigma(m)] & \text{si la factorisation } lm \text{ est standard.} \end{array}$$

**Exemple:** $a^2b^2 \xrightarrow{\sigma} [a,[[a,b],b]]$ et non pas $[[a,[a,b]],b]$.

### 3.3.4 Factorisation d'un mot

**Théorème 3.3.2** *Tout mot $w$ de $X^*$ se décompose de façon unique en un produit décroissant de mots de Lyndon i.e.*

$$w = l_1 l_2 \cdots l_n \quad \textit{avec} \quad l_1 \geq l_2 \geq \cdots \geq l_n.$$

**Preuve:** voir Viennot [34]. □

#### 3.3.4.1 Algorithme de factorisation

Cet algorithme (voir [21] construit progressivement la liste $ll$ des facteurs de $w$ en partant des lettres de $w$ et s'arrête lorsque $ll$ est décroissante au sens large.

**Definition 3.3.4** *Une liste de crochets de Lyndon* $ll = x_1x_2\cdots x_n$ *est dite* standard ssi $\forall i, (i = 1..n)$ *l'assertion suivante est vraie:*

- *Soit* $x_i$ *est une lettre,*
- *Soit* $x_i = [l, m]$ *avec* $m \geq x_j\ \forall j > i$.

**Remarque:** une liste formée uniquement de lettres est standard de même que n'importe quelle liste décroissante au sens large.

Factorisation ($w$: Mot): ListeDeCrochetsDeLyndon ==
  Initialisation:
    $ll$:ListeDeCrochetsDeLyndon := liste des lettres de $w$
  tq $ll$ n'est pas décroissante au sens large faire
    • Repérer la dernière inversion $x_i < x_{i+1}$ de la liste $ll$
    • Crocheter $x_i$ et $x_{i+1}$
  ftq
  *retourner*($ll$)

**Invariant du "tant que":**
La liste $ll = x_1x_2\cdots x_n$ est une liste *standard.* Cet invariant garantit que l'opération de crochetage construit bien des crochets de Lyndon.

**Exemple:** Soit $w = ba^2b^2a$ ; on a:

$$\begin{array}{lcccccc} ll_0 = & b & a & a & b & b & a \\ ll_1 = & b & a & [a & b] & b & a \\ ll_2 = & b & a & [[a & b] & b] & a \\ ll_3 = & b & [a & [[a & b] & b]] & a \end{array}$$

On obtient finalement un produit de trois mots de Lyndon: $w = b(a^2b^2)a$.

### 3.3.5 Implantation en Scratchpad

J'ai choisi de représenter les mots de Lyndon sous forme arborescente en les confondant, du point de vue de leur codage interne, avec leur forme parenthésée. Il convient donc d'implanter en premier la gestion de $\mathcal{M}agma\langle X\rangle$.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++ Gestion des magmas:
++           un magma est un ensemble de mots parentheses construits
++           sur un ensemble ordonne quelconque.
++
++ 3 ordres sont implantes:
++
++   1) L'ordre recursif est l'ordre habituel sur les structures d'arbres:
++         comparaison des sous-arbres gauche
++         puis (en cas d'egalite) des sous-arbres droit
```

```
++   2) L'ordre lexicographique: (du dictionnaire)
++   3) Recursif par longueur
++
++ Version 1 du 27 dec. 1990
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- Revisee 20/07/91  V4.3

)abbrev domain MAGMA Magma

Magma(VarSet:OrderedSet):Public == Private where
   Public == Join(OrderedSet,RetractableTo VarSet) with
      "*":($,$) -> $
      left: $ -> $                    -- facteur gauche
      right: $ -> $                   -- facteur droit
      first: $ -> VarSet              -- premiere lettre
      rest : $ -> $                   -- sauf premiere lettre
      length: $ -> PositiveInteger
      coerce: $ -> OrderedFreeMonoid1(VarSet)
      coerce: $ -> OutputForm
      coerce: $ -> Expression
      recursif: ($,$) -> Boolean
      lexico  : ($,$) -> Boolean
      mirror  : $ -> $                -- image miroir
      varList : $ -> List VarSet      -- liste des lettres

    -- attributes
      orderPreserve("*")

   Private == add
    -- representation
      VWORD := Record(left:$ ,right:$)
      Rep:= Union(VarSet,VWORD)
```

L'implantation des mots de Lyndon se fait par héritage du constructeur **Magma** :

```
)abbrev domain LWORD LyndonWord

LyndonWord(VarSet:OrderedSet):Public == Private where
   OFMON ==> OrderedFreeMonoid1(VarSet)
   PI    ==> PositiveInteger
   NNI   ==> NonNegativeInteger
   I     ==> Integer
   OF    ==> OutputForm

   Public == Join(OrderedSet,RetractableTo VarSet) with
      left:       $ -> $
      right:      $ -> $
      length:     $ -> PI
      lexico: ($,$) -> Boolean      -- ordre lexicographique
```

```
    coerce:      $ -> OFMON
    coerce:      $ -> OutputForm
    coerce:      $ -> Expression
    coerce:      $ -> Magma VarSet
    factor:      OFMON -> List $  -- factorisation decroissante
    lyndon?:     OFMON -> Boolean
    lyndon:      OFMON -> $
    lyndonIfCan:OFMON -> Union($, "failed")
    varList:     $ -> List VarSet

  -- construction des mots de Lyndon de longueur bornee
    LyndonWordsList1: (List VarSet, PI)  -> Vector List $
    LyndonWordsList : (List VarSet, PI)  -> List $

Private == Magma(VarSet) add
  -- Representation
    Rep:= Magma(VarSet)
  ...
```

L'agorithme de génération des mots de Lyndon de longueur bornée découle directement de la définition 3.3.3 . Il convient de noter que la fonction de comparaison des crochets de Lyndon est assez coûteuse car l'ordre le plus naturel sur $\mathcal{M}agma\langle X\rangle$ à savoir :

$$[x_1, y_1] < [x_2, y_2] \text{ ssi } x_1 < x_2 \text{ ou } (x_1 = x_2 \text{ et } y_1 < y_2)$$

n'est pas compatible avec l'ordre lexicographique sur les mots de $X^*$ comme le montre l'exemple sur les deux mots de Lyndon $[[a, b], b]$ et $[a, [b, c]]$.

## 3.4 Les polynômes de Lie

### 3.4.1 La représentation par les polynômes non commutatifs

On peut représenter les polynômes de Lie, sous leur forme développée, comme de simples polynômes en variables non commutatives. Cette représentation est suffisante pour la plupart des calculs mais a l'inconvénient de prendre beaucoup de place en mémoire.

$$\begin{aligned} [a, b] &= ab - ba \\ [a[a, b]] &= a^2b - 2aba + ba^2 \end{aligned}$$

J'ai implanté les polynômes de Lie de trois manières

1. en utilisant la base de Lyndon, (voir 3.4.3)
2. sous forme développée en représentation distribuée, et
3. sous forme développée en représentation récursive.

L'avantage est de pouvoir lancer les mêmes calculs en utilisant successivement chacune des trois implantations, de comparer les résultats obtenus et de ...corriger les éventuels "bugs".

Voici le début de l'implantation du constructeur **LiePolynomial1** gérant les polynômes de Lie sous forme développée (représentation récursive).

```
)abbrev domain LPOLY1 LiePolynomial1

LiePolynomial1(VarSet:OrderedSet, K:Ring) :Public == Private where
   XRPOLY  ==> XRecursivePolynomial(VarSet,K)
   XDPOLY  ==> XDistribuedPolynomial(VarSet,K)
   LWORD   ==> LyndonWord(VarSet)
   NNI     ==> NonNegativeInteger
   RN      ==> RationalNumber
   XEXPPKG ==> XExponentialPackage(K, VarSet, XRPOLY)

   Public == FreeLieAlgebra(VarSet, K) with
      if K has Module(RN) then
        exp: ($, NNI) -> XRPOLY  -- exponentielle tronquee d'un poly. de Lie
        Hausdorff:($,$,NNI) -> $ -- serie de Hausdorff tronquee

   Private ==  XRPOLY add
      --representation
        Rep := XRPOLY

      --definition
        if K has Module(RN) then
          exp (p,n) == exp(p::Rep, n)$XEXPPKG
          Hausdorff(x,y,n) == log(exp(x,n) * exp(y,n) , n)$XEXPPKG

        construct(x,y) == x *$Rep y - y *$Rep x
        .....
```

### 3.4.2 La représentation dans la base de Lyndon

Un polynôme $p \in \mathcal{L}ie_K\langle X\rangle$ est considéré comme une combinaison linéaire de crochets de Lyndon.

$$p = \sum_{i=1}^{n} \alpha_i [l_i] \quad \text{avec } \alpha_i \in K.$$

#### 3.4.2.1 Calcul du crochet de Lie de deux polynômes

Le problème est résolu si l'on sait exprimer le crochet de Lie de 2 crochets de Lyndon $u$ et $v$ comme une combinaison linéaire de crochets de Lyndon.

La fonction **CrochetDeLie** est conçue pour se ramener au cas où $u < v$ :

```
CrochetDeLie (u, v: CrochetsDeLyndon): PolynômeDeLie ==
  si u = v alors
    retourner(0)
  sinon
    si u < v alors
      retourner(crochet(u, v))
    sinon
      retourner(− crochet(v, u))
    fsi
  fsi
```

Le mot $uv$ est alors un mot de Lyndon ; la difficulté tient au fait que la factorisation $uv$ n'est pas forcément la factorisation standard (voir 3.3.3.2).

Posons $u = [u_0, u_1]$ ; si $u_1 \geq v$ alors la factorisation $uv$ est standard et le problème est résolu. Dans le cas contraire, on utilise l'identité de JACOBI:

$$\begin{aligned}[u, v] &= [[u_0, u_1]v] \\ &= \underbrace{[u_0[u_1, v]]}_{x} + \underbrace{[[u_0, v]u_1]}_{y}\end{aligned}$$

Par hypothèse, on a:

$$u_0 < u_0u_1 < u_1 < v.$$

La factorisation du mot de Lyndon $x = u_0u_1v$ n'est pas forcément standard, mais on est passé de $[[u_0, u_1]v]$ à $[u_0[u_1, v]]$ ; la longueur du facteur gauche a donc strictement diminué.

Dans $y$, on sait que $u_0v$ est un mot de Lyndon, ce qui nous ramène au problème de départ à ceci près que:

$$[[u_0, u_1]v] < [[u_0, v]u_1]$$

pour l'ordre lexicographique.

L'algorithme suivant termine:

```
crochet (u, v: CrochetsDeLyndon): PolynômeDeLie ==
  − on a: u < v
  si u est une lettre alors
    retourner([u, v])
  fsi
  posons u = [u_0, u_1]
  si u_1 ≥ v alors
    retourner([u, v])
  sinon
    x := crochet(u_0, crochet(u_1, v))
    y := CrochetDeLie(crochet(u_0, v),u_1)
    retourner(x + y)
  fsi
```

Les appels récursifs peuvent être vus comme un système de réécriture formé des règles:

$$\begin{cases} [[u_0,u_1]v] & \longrightarrow [u_0[u_1,v]] & \text{si } u_1 < v & (x) \\ [[u_0,u_1]v] & \longrightarrow [[u_0,v]u_1] & \text{si } u_1 < v & (y) \\ [u,v] & \longrightarrow [v,u] & \text{si } u > v & (z) \end{cases}$$

Les règles $y$ et $z$ ne peuvent s'appliquer qu'un nombre fini de fois car, interprétées sur des mots de $X^*$, elles transforment un mot en un autre mot formé des mêmes lettres mais plus grand pour l'ordre lexicographique.

Entre deux réécritures $y$ ou $z$, la règle $x$ ne peut s'appliquer qu'un nombre fini de fois car elle fait décroître strictement la longueur du facteur gauche.

#### 3.4.2.2 Calcul du miroir d'un polynôme de Lie

Par définition, le miroir $\tilde{p}$ d'un polynôme $p$ est obtenu en prenant le miroir de chacun de ses mots.

**Exemple :**

$$\begin{array}{lrcl} \text{Soit} & p & = & [a,b]+[a[a,b]] \\ & & = & ab-ba+a^2b-2aba+ba^2 \\ \text{on a:} & \tilde{p} & = & ba-ab+ba^2-2aba+a^2b \\ & & = & -[a,b]+[a[a,b]] \end{array}$$

**Proposition 3.4.1** *Le miroir d'un polynôme de Lie est obtenu par simple changement de signe de ses crochets de longueur paire ie.*

$$p=\sum_i \alpha_i[l_i] \implies \tilde{p}=\sum_i -(-1)^{|l_i|}\alpha_i[l_i]$$

### 3.4.3 Implantation en SCRATCHPAD

Il s'agit de représenter un polynôme de Lie comme combinaison linéaire de crochets de Lyndon. L'implantation des opérations linéaires est héritée du constructeur de domaine `FreeModule`.

```
)abbrev domain LPOLY LiePolynomial

LiePolynomial(VarSet:OrderedSet, K:Ring) : Public == Private where
   LWORD   ==> LyndonWord(VarSet)
   WORD    ==> OrderedFreeMonoid1(VarSet)
   XDPOLY  ==> XDistribuedPolynomial(VarSet,K)
   XRPOLY  ==> XRecursivePolynomial(VarSet,K)
   Term    ==> Record(k: LWORD, c: K)

   Public == FreeLieAlgebra(VarSet,K) with
      ListOfTerms : $ -> List Term
```

```
    leadingTerm : $ -> LWORD
    leadingCoef : $ -> K
    reductum : $ -> $
    monom : (LWORD,K) -> $
    LiePolyIfCan: XDPOLY -> Union($, "failed")
    construct: (LWORD, LWORD) -> $      -- crochet de Lie
    construct: (LWORD, $) -> $
    construct: ($, LWORD) -> $

Private == FreeModule(K, LWORD) add
     import(Term)

    --representation
      Rep := List Term
      .....
```

### 3.4.3.1 Identification d'un polynôme de Lie

Etant donné un polynôme en variables non commutatives, existe–t–il un algorithme simple pour décider si c'est un polynôme de Lie ? Si oui, comment calculer sa décomposition en crochets de Lyndon ?

Nous verrons que de ce point de vue, l'utilisation de la base de Lyndon est préférable.

Soit $l$ un mot de Lyndon quelconque,
$[l]$ le polynôme de Lie correspondant.

**Lemme 3.4.1** *$l$ est le plus petit mot apparaissant dans $[l]$ et son coefficient est égal à 1 i.e.*

$$[l] = l + \sum_{i \in I} \alpha_i w_i \quad avec \quad \forall i \in I,\ \alpha_i \in K,\ w_i \in X^*,\ w_i > l$$

**Preuve:** voir [20].

Ce lemme est particulier à la base de Lyndon et n'a pas d'équivalent pour la base de Hall.

**Remarque:** Il se peut que certains mots $w_i$ soient eux–aussi des mots de Lyndon:

$$[a,[b,c]] = abc - acb - bca + cba.$$

le mot $acb$ étant lui-aussi un mot de Lyndon.

**Corollaire 3.4.1** *Le plus petit mot d'un polynôme de Lie est un mot de Lyndon.*

**Preuve:** Il suffit de considérer le plus petit crochet de Lyndon intervenant dans la décomposition du polynôme de Lie. □

Soit un polynôme $p$ en représentation distribuée ; l'algorithme suivant décompose $p$ dans la base de Lyndon lorsque celui-ci est un polynôme de Lie.

```
conversion (p:PolynomeDistribue): PolynomeDeLie ==
  Initialisation:
    r: PolynomeDeLie := 0;
    p1: PolynomeDistribue := p
  tq p1 ≠ 0 faire
    Soit c.w le plus petit monome de p1
    si w n'est pas un mot de Lyndon alors
      echec
    sinon
      r := r + c[w] ; p1 := p1 − c[w]
    fsi
  ftq
  retourner(r)
```

On observe que cet algorithme est plus efficace si le plus petit monôme est en tête[1] de la liste représentant les polynômes.

## 3.5 L'algèbre enveloppante

### 3.5.1 Définition

Rappelons que l'algèbre enveloppante [6] d'une algèbre de Lie $\mathcal{L}$ est définie comme le quotient $\mathcal{U}$ de l'algèbre tensorielle

$$\mathcal{T} = \mathcal{L}^0 \oplus \mathcal{L}^1 \oplus \cdots \oplus \mathcal{L}^n \oplus \cdots \qquad \text{avec } \mathcal{L}^0 = K.1 \text{ et } \mathcal{L}^n = \mathcal{L} \otimes \mathcal{L} \otimes \cdots \otimes \mathcal{L} \quad (n \text{ facteurs}).$$

par l'idéal $\mathcal{J}$ engendré par les éléments de la forme

$$x \otimes y - y \otimes x - [x, y] \quad \text{pour } x, y \in \mathcal{L}$$

On a donc: $\mathcal{U} = \mathcal{T}/\mathcal{J}$.

**Théorème 3.5.1** *L'algèbre enveloppante de l'algèbre de Lie libre $\mathcal{L}ie_K\langle X\rangle$ s'identifie à l'algèbre des polynômes non commutatifs $K\langle X\rangle$.*

**Preuve:** Voir [15]

### 3.5.2 La base PBWL

**Théorème 3.5.2 (Poincaré-Birkoff-Witt)**
*Si une algèbre de Lie $\mathcal{L}$ admet une base $\{P_1, P_2, \ldots, P_n, \ldots\}$ alors son algèbre enveloppante admet comme base les éléments de la forme $P_{i_1} P_{i_2} \cdots P_{i_n}$ avec $P_{i_1} \geq P_{i_2} \geq \cdots \geq P_{i_n}$ et $n \geq 0$.*

Ce théorème appliqué sur la base de Lyndon de l'algèbre de Lie libre nous fournit une base de l'algèbre $K\langle X\rangle$ , base que nous noterons $PBWL$.

[1] et non en queue comme c'est souvent le cas.

**Remarque:** Il existe une bijection entre les deux bases $X^*$ et $PBWL$.

Soit $w \in X^*$; considérons la factorisation en produit décroissant de mots de Lyndon:

$$w = l_1 l_2 \cdots l_n \quad \text{avec } l_1 \geq l_2 \cdots \geq l_n.$$

Il lui correspond alors un élément $Q_w$ de la base $PBWL$ défini par

$$Q_w = [l_1][l_2] \cdots [l_n].$$

où $[l_1], [l_2], \cdots, [l_n]$ désignent les polynômes de Lie associés aux mots de Lyndon.

### 3.5.3 Décomposition d'un mot dans la base PBWL

L'algorithme [21] est basé sur la formule:

$$ab = [a, b] + ba$$

qui est la décomposition du mot $ab$ dans la base $PBWL$.

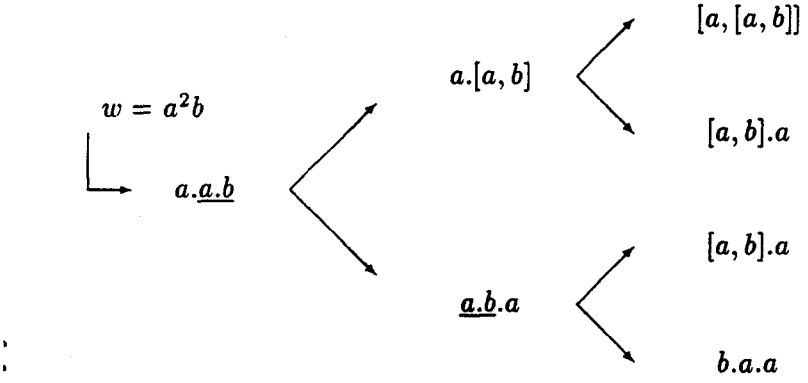
**Lemme 3.5.1** *Soit $ll = \{x_1, x_2, \cdots, x_n\}$ une liste standard (voir 3.3.4) représentant un produit de crochets de Lyndon. Soit $(x_i, x_{i+1})$ la dernière inversion $x_i < x_{i+1}$ de $ll$. Alors les listes*

$$\begin{cases} ll_0 = \{x_1, \ldots, [x_i, x_{i+1}], \ldots, x_n\} \\ ll_1 = \{x_1, \ldots, x_{i+1}, x_i, \ldots, x_n\} \end{cases}$$

*sont standards.*

L'algorithme de décomposition d'un mot $w$ peut être vu comme un système de réécriture $ll \longrightarrow ll_0 + ll_1$ où la liste $ll$ initiale est formée des lettres de $w$.

**Exemple** Soit $X = \{a, b\}$ un alphabet. Soit à décomposer le mot $w = a^2b$ dans la base $PBWL$. Le déroulement de l'algorithme est le suivant :




Ainsi, $w = [a, [a, b]] + 2[a, b].a + ba^2$.

**Proposition 3.5.1** *Le passage de la base $X^*$ à la base $PBWL$ se fait sans utiliser de coefficients fractionnaires. La conversion d'un mot $w$ se fait en utilisant uniquement des éléments $Q_u$ tels que $|u| = |w|$ et $u \geq w$ i.e.*

$$w = Q_w + \sum_i \alpha_i Q_{u_i}, \quad \forall i,\ |u_i| = |w| \text{ et } u_i \geq w$$

### 3.5.4 Les polynômes dans la base de PBWL

L'idée est de représenter les polynômes de $K\langle X\rangle$ dans la base de $PBWL$ et de maintenir cette représentation pour les opérations de somme et de produit.

#### 3.5.4.1 Produit de deux polynômes

Le problème est résolu si l'on sait multiplier deux éléments de la base de $PBWL$ et exprimer le produit dans cette même base.

Idée de l'algorithme:
Soient $u = p_1 p_2 \dots p_i$ et $v = q_1 q_2 \dots q_j$ deux éléments de la base $PBWL$.
On a:

$$uv = p_1 \dots p_i \bullet q_1 \dots q_j$$

Si $p_i \geq q_1$, c'est fini; sinon on effectue la réécriture:

$$\begin{aligned} uv = \quad & p_1 \dots p_{i-1} \bullet [p_i, q_1] \bullet q_2 \dots q_j \\ + \quad & p_1 \dots p_{i-1} \bullet q_1\ p_i \bullet q_2 \dots q_j \end{aligned}$$

Il convient de traiter le crochet $[p_i, q_1]$ car cette factorisation n'est pas forcément standard.

Les $\bullet$ signalent les endroits où l'ordre des facteurs est peut-être incorrect. Il faut donc relancer récursivement l'algorithme pour traiter les inversions éventuelles. Il est assez facile de prouver que ce processus termine. Cependant, l'algorithme doit être programmé avec soin si l'on souhaite limiter le nombre de comparaisons à effectuer.

#### 3.5.4.2 Expression d'un polynôme distribué dans la base $PBWL$

Le problème consiste à calculer la décomposition dans la base $PBWL$ d'un polynôme $p$ exprimé comme une combinaison linéaire de mots de $X^*$.

Il est bien sûr possible de convertir séparément chaque mot de $p$ en utilisant l'algorithme de la section 3.5.3 ; un autre algorithme inspiré de (3.4.3.1) a cependant ma préférence.

**Lemme 3.5.2** *$w$ est le plus petit mot du polynôme $Q_w$ i.e.*

$$Q_w = w + \sum_i \alpha_i u_i \quad \text{avec } \forall i,\ u_i > w, \quad \alpha_i \in K$$

**Preuve:** assez facile à partir du lemme 3.4.1. □

Ce lemme justifie l'algorithme suivant:

conversion ($p$:PolynomeDistribue): PolynomeDansPBWL ==
  Initialisation:
    $r$: PolynomeDansPBWL := 0;
    $p1$: PolynomeDistribue := $p$
  tq $p1 \neq 0$ faire
    Soit $c.w$ le plus petit monome de $p$
    Décomposer $w$ en produit décroissant de mots de Lyndon
    $r := r + cQ_w$ ; – calculs dans la base $PBWL$
    $p1 := p1 - cQ_w$ ; – calculs dans la base $X^*$
  ftq
  *retourner*($r$)

### 3.5.5 Implantation en Scratchpad

Il convient d'abord de gérer les éléments de la base $PBWL$ ; ceux-ci sont représentés par des listes décroissantes de mots de Lyndon.

```
)abbrev domain PBWLB PoincareBirkoffWittLyndonBasis

PoincareBirkoffWittLyndonBasis(VarSet: OrderedSet): Public == Private where
   WORD    ==> OrderedFreeMonoid1(VarSet)
   LWORD   ==> LyndonWord(VarSet)
   LWORDS  ==> List(LWORD)
   NNI     ==> NonNegativeInteger

   Public == Join(OrderedSet, RetractableTo LWORD) with
      coerce: VarSet -> $
      length: $ -> NNI
      ListOfTerms: $ -> LWORDS
      coerce: $ -> WORD
      varList: $ -> List VarSet
      first: $ -> LWORD
      rest: $ -> $
      1: () -> $

   Private == add
    -- Representation
     Rep := LWORDS
```

Les polynômes sont repésentés par des combinaisons linéaires de ces éléments de base ; on notera l'utilisation du constructeur de domaine **FreeModule**, ce qui évite d'avoir à reprogrammer les opérations linéaires comme la somme ou la différence de deux polynômes.

```
)abbrev domain XPBWPOLY XPBWPolynomial

XPBWPolynomial(VarSet:OrderedSet,R:Ring): XDPcat == XDPdef where

  WORD   ==> OrderedFreeMonoid1(VarSet)
```

```
LWORD  ==> LyndonWord(VarSet)
LWORDS ==> List LWORD
BASIS  ==> PoincareBirkoffWittLyndonBasis(VarSet)
TERM   ==> Record(k:BASIS, c:R)
LTERMS ==> List(TERM)
LPOLY  ==> LiePolynomial(VarSet,R)
XDPOLY ==> XDistribuedPolynomial(VarSet,R)
TERM1  ==> Record(k:LWORD, c:R)
NNI    ==> NonNegativeInteger
RN     ==> RationalNumber

XDPcat == XPolynomialsCat(VarSet,R) with
  leadingTerm : $ -> BASIS
  leadingCoef : $ -> R
  reductum    : $ -> $
  coerce      : LPOLY -> $
  ListOfTerms : $ -> LTERMS
  monom       : (BASIS, R) -> $
  coerce      : $ -> XDPOLY
  LiePolyIfCan: $ -> Union(LPOLY,"failed")
  product     : ($,$,NNI) -> $          -- produit tronque a l'ordre n

  if R has Module(RN) then
     exp      : ($,NNI) -> $            -- calcul tronque a l'ordre n
     log      : ($,NNI) -> $

XDPdef == FreeModule(R,BASIS) add
  -- Representation
     Rep:= LTERMS
```

### 3.5.6 Formules de Backer-Campbell-Hausdorff

Les calculs des fonctions *exponentielle* et *logarithme* sont tronqués à un ordre quelconque précisé par l'utilisateur et s'effectuent dans l'algèbre enveloppante en appliquant les formules classiques (on suppose $p$ sans terme constant):

$$\exp(p) = 1 + p + \frac{p^2}{2!} + \cdots$$

$$\log(1+p) = p - \frac{p^2}{2} + \frac{p^3}{3} + \cdots$$

On obtient ainsi le produit de Hausdorff à l'ordre 4:

$$\begin{aligned} a_H b &= \log(e^a e^b) \\ &= a + b + \frac{1}{2}[ab] + \frac{1}{12}[a^2 b] + \frac{1}{12}[ab^2] + \frac{1}{24}[a^2 b^2] + \cdots \end{aligned}$$

Le choix de la base $PBWL$ facilite l'identification de la série tronquée $\log(e^a e^b)$ comme un polynôme de Lie. Ce choix est raisonnablement efficace mais n'est certainement pas optimum si l'on vise à calculer uniquement les formules de Backer-Campbell-Hausdorff.

## 3.6 Conclusion

On aurait pu utiliser la base de Ph. Hall [34] notée $\mathcal{H}all\langle X\rangle$ ; celle-ci est définie récursivement en partant d'une relation d'ordre sur $\mathcal{M}agma\langle X\rangle$:

1. Les lettres de $X$ sont des crochets de Hall.
2. Si $a$ est une lettre et si $b$ est un crochet de Hall, alors $[a,b]$ est un crochet de Hall ssi $a < b$.
3. Si $a$ et $[b,c]$ sont des crochets de Hall, alors $[a,[b,c]]$ est un crochet de Hall **ssi** $b \leq a < [b,c]$.

L'ordre sur les crochets doit seulement vérifier la condition (Meier-Wunderli 52) :

$$\forall a,b \in \mathcal{H}all\langle X\rangle, \quad a < [a,b] \text{ et } b < [a,b].$$

Il est clair que l'ordre lexicographique par longueur convient mais ce n'est pas le seul. Cet ordre peut même être construit dynamiquement en même temps que l'on construit la base $\mathcal{H}all\langle X\rangle$.

On constate que cette définition est très semblable à la définition récursive des crochets de Lyndon à l'exception de l'ordre qui, pour la base de Lyndon, est rigoureusement imposé (ordre lexicographique).

Les propriétés mentionnées dans ce chapitre ont toutes leur équivalent pour les bases de Hall sauf celles liées au lemme 3.4.1:

$$[l] = l + \sum_{i\in I} \alpha_i w_i \quad \text{avec } \forall i \in I,\ \alpha_i \in K,\ w_i \in X^*,\ w_i > l$$

Les algorithmes de conversion des sections 3.4.3.1 et 3.5.4.2 qui découlent directement de ce lemme ne sont donc plus applicables.

# 4

# Bases-standard

# Une formulation générale à la "Knuth-Bendix"

## 4.1 Introduction

Depuis de nombreuses années, les mathématiciens utilisent la réécriture algébrique pour obtenir des critères effectifs d'égalité entre des espaces vectoriels (resp. des idéaux) de polynômes définis par un nombre fini de générateurs.

Le calcul des bases-standard[1] fut initié par Buchberger [4] en 1970 pour étudier les idéaux de polynômes commutatifs ; ce calcul repose sur les concepts qui soustendent l'algorithme de Knuth-Bendix [17]. Il s'agit ici de développer ce point de vue et d'adapter aux polynômes les calculs que Knuth effectue sur les arbres. Les techniques exposées sont utilisables aussi bien en algèbre commutative que non commutative, en algèbre différentielle ou dans les algèbres de Lie. Cette présentation unifiée est obtenue en introduisant le concept de *filtrage* qui est une relation d'ordre partiel sur l'ensemble des mots utilisés.

En ce qui concerne la réécriture en général, on pourra consulter l'article de N.Dershowitz et J.P. Jouannaud [5] ; la réécriture en algèbre commutative, appliquée en automatique au problème de l'identifiabilité structurelle globale, a été étudiée dans la thèse de F. Ollivier [23]. F. Mora dans [22] traite le cas des polynômes non commutatifs.

Le calcul des bases de Groebner est réputé utiliser beaucoup de place mémoire car le nombre de règles peut devenir grand à certaines étapes de l'algorithme.

Cette difficulté n'apparait pas lorsque l'on traite des espaces vectoriels ou des idéaux à droite de polynômes non commutatifs. Dans ces deux cas, le nombre de règles ne peut que diminuer mais jamais augmenter ; il est donc borné par le nombre de générateurs donnés au départ.

Pour ce qui nous concerne, ces deux cas ont un intérêt pratique certain ; les bases-standard d'espaces vectoriels permettent de traiter élégemment le calcul de rang, somme et intersection d'espaces vectoriels. Elles permettent déterminer l'image et le noyau d'une application linéaire[2] sans utiliser les matrices.

Quant aux bases-standard d'idéaux à droite, elles permettent de représenter sous forme canonique les séries rationnelles en variables non commutatives introduites par M.P.Schützenberger. Ceci constitue donc une alternative à la représentation matricielle

[1] Buchberger les appela bases de Gröbner.

[2] On passe l'application linéaire $f$ en paramètre au programme calculant $Ker(f)$.

qui, même losqu'elle est minimale, est unique ...à un changement de base près.

## 4.2 Définitions de base et problématique

Knuth a étudié la réécriture dans des algèbres *généralisées*. Celles-ci sont formées d'arbres n-aires pouvant éventuellement comporter des variables dans les feuilles. Quoique l'on puisse représenter les polynômes par de tels arbres, nous préférons considérer les polynômes comme des combinaisons linéaires de *mots*.

D'une façon générale, nous nous plaçons dans une algèbre de polynômes $\mathcal{A} = K[\mathcal{W}]$ , définie sur un corps $K$ et un ensemble de mots $\mathcal{W}$ totalement ordonné. [3]

On considère une *relation d'équivalence* notée $\equiv_E$ compatible avec l'addition et la multiplication par un élément de $K$. Cette relation est équivalente à la donnée d'un sous-espace vectoriel $E$ de $\mathcal{A}$.

On considère qu' un *système de réécriture* $\mathcal{R}_G$ est défini par

1. un ordre total sur les mots noté $<$.
2. un ensemble fini de $n$ *règles* de la forme $w_i \rightarrow q_i$ où $w_i$ est un mot et $q_i$ un polynôme. On suppose que $w_i$ est strictement supérieur à tous les mots de $q_i$. Cela revient à se donner un ensemble $G$ de $n$ polynômes :

$$G = \{g_i = w_i - q_i |\ 1 \leq i \leq n\}$$

3. une technique de *filtrage* précisant de quelle manière on effectue les substitutions.

Nous verrons que le filtrage suppose la définition d'une deuxième relation d'ordre (partiel) sur les mots que nous noterons $\preceq$ et dont la sémantique est:

*une règle* $u \longrightarrow q$ *est applicable pour réécrire le mot* $v$ **ssi** $u \preceq v$.

On supposera toujours que pour deux mots quelconques u,v :

$u \preceq v \Longrightarrow u \leq v$

On notera :

- $p \xrightarrow{*} q$ pour signifier que $q$ se déduit de $p$ par l'application d'un nombre quelconque de pas de réécriture.
- $p \xrightarrow{+} q$ pour signifier que $q$ se déduit de $p$ par l'application d'un nombre non nul de pas de réécriture.

On supposera toujours que le système $\mathcal{R}_G$ est **cohérent** par rapport à la relation d'équivalence c'est-à-dire que

$p \xrightarrow{*} q \Longrightarrow p \equiv_E q$

[3] Pour l'étude des polynômes de Lie, $\mathcal{W}$ peut être la base de Lyndon.

ce qui revient à supposer que $G \subset E$.

Un polynôme est dit *réduit* pour le système $\mathcal{R}$ si aucune règle de $\mathcal{R}$ ne peut lui être appliquée.

On appelle *forme normale* d'un polynôme $p$, un polynôme $\overline{p}$ réduit tel que $p \xrightarrow{*} \overline{p}$ .

Un système de réécriture *reconnait* une relation d'équivalence **ssi** :

1. Tout polynôme $p$ admet une forme normale unique $\overline{p}$.
2. Deux polynômes quelconques sont équivalents **ssi** ils ont même forme normale autrement dit:

$$\forall p_1, p_2 \in \mathcal{A}, \quad p_1 \equiv_E p_2 \Longleftrightarrow \overline{p_1} = \overline{p_2}$$

La question est la suivante :

> Une relation d'équivalence $\equiv_E$ étant donnée, construire un système $\mathcal{R}_G$ qui la reconnait. $G$ est alors appelée une *base-standard* de $E$. On obtient ainsi un algorithme pour tester facilement l'équivalence de deux polynômes.

### 4.2.1 Exemple

Soit l'alphabet $X = \{x, y\}$. Considérons l'algèbre $\mathcal{A} = \mathbb{Q}\langle X \rangle$ et les deux règles :

$$\begin{cases} x & \longrightarrow 1 \quad (r_1) \\ y^2 & \longrightarrow 1 \quad (r_2) \end{cases}$$

l'ordre sur les mots de $X^*$ étant l'ordre lexicographique par longueur.

> **Par modification de la technique de filtrage, on obtient divers systèmes de réécriture qu'il faut considérer comme distincts.**

Soit à calculer, dans chacun des cas, la forme normale du polynôme $p = 2y^2 + xy + yx$.

- **Filtrage 1** : Etude de l'espace vectoriel engendré par les polynômes:

  $g_1 = x - 1$ et $g_2 = y^2 - 1$

  Les substitutions se font par mots entiers c'est-à-dire

  $\forall u, v \in X^* \quad u \preceq v$ **ssi** $u = v$

  On a: $p \xrightarrow{r_2} 2 + xy + yx$ qui est une forme normale.

- **Filtrage 2** : Etude de l'idéal à droite engendré par $g_1$ et $g_2$:
  Les substitutions se font à gauche des mots c'est-à-dire

  $\forall u, v \in X^* \quad u \preceq v$ **ssi** $u$ est un facteur gauche de $v$

  On a: $p \xrightarrow{r_2} 2 + xy + yx \xrightarrow{r_1} 2 + y + yx$ qui est une forme normale.

- **Filtrage 3** : Etude de l'idéal bilatère engendré par $g_1$ et $g_2$ :

  $\forall u, v \in X^* \quad u \preceq v$ **ssi** $u$ est un facteur de $v$

  On a: $p \xrightarrow{r_2} 2 + xy + yx \xrightarrow{r_1} 2 + 2y$ qui est une forme normale.

## 4.3 L'arrêt des calculs

**Définition 4.3.1** *Un système de réécriture est dit* noethérien **ssi** *toutes les chaines de réécriture sont finies.*

**Proposition 4.3.1** *Pour un système noethérien, la relation* $\xrightarrow{+}$ *est une relation d'ordre (partiel) sur les polynômes.*

**Preuve:** la relation $x \xrightarrow{+} y$ est transitive. La situation $x \xrightarrow{+} y$ et $y \xrightarrow{+} x$ est impossible car l'existence d'une boucle contredit la noethérianité. □

Nous avons dit que le calcul que le calcul d'une base–standard présuppose le choix d'un ordre total (noté $<$) sur les mots.

**Notations**

Le plus grand mot d'un polynôme $p$ est alors appelé son *terme dominant* noté $\mathrm{lt}(p)$.
Le coefficient de ce mot est appelé *coefficient dominant* de $p$ et noté $\mathrm{lc}(p)$.
Le polynôme $p$ privé de son monôme de tête est noté $\mathrm{rest}(p)$

On impose que toutes les règles de réécriture soient de la forme $w \to q$ avec $w > \mathrm{lt}(q)$.

### 4.3.1 Ordre admissible

L'ordre total sur les mots est prolongé en un ordre partiel sur les polynômes:

$$\begin{array}{lll} p < q & \text{si} & p = 0 \text{ et } q \neq 0 \\ & \text{ou si} & \mathrm{lt}(p) < \mathrm{lt}(q) \\ & \text{ou si} & \mathrm{lt}(p) = \mathrm{lt}(q) \text{ et } \mathrm{rest}(p) < \mathrm{rest}(q). \end{array}$$

Un ordre est dit *bien fondé* **ssi** toute chaîne strictement décroissante est finie.

**Lemme 4.3.1** *Si l'ordre sur les mots est bien fondé, alors l'ordre sur les polynômes est bien fondé.*

Un ordre bien fondé sur les mots est dit *admissible* **ssi** l'ordre qui en résulte sur les polynômes vérifie la propriété:

$$p \xrightarrow{+} q \Longrightarrow p > q$$

L'existence d'un ordre admissible sur les mots entraîne que le système est noethérien.

#### 4.3.1.1 Exemple

Considérons le système de réécriture

$$\begin{cases} ab \longrightarrow a & (r_1) \\ ac \longrightarrow abc & (r_2) \end{cases}$$

$\forall u, v \in (a+b+c)^*$, $u \preceq v$ **ssi** $u$ est un facteur gauche de $v$.

**L' ordre lexicographique n'est pas admissible** car on obtient la chaîne infinie :

$abc \xrightarrow{r_1} ac \xrightarrow{r_2} abc \cdots$

La difficulté tient au fait que l'ordre lexicographique n'est pas compatible avec la multiplication à droite[4] ; l'implication

$\forall u, v, x \in (a+b+c)^* \quad u < v \Longrightarrow ux < vx$

est fausse.

On vérifie sans peine que l'ordre lexicographique par longueur ne présente pas cet inconvénient.

## 4.4 La confluence

**Definition 4.4.1** *Un système de réécriture est dit* confluent **ssi**

$$\forall u_0, p, q \in \mathcal{A} \quad \textit{tels que} \quad \begin{cases} u_0 \xrightarrow{*} p \\ u_0 \xrightarrow{*} q \end{cases}$$

$$\exists u_1 \in \mathcal{A} \quad \textit{tel que} \quad \begin{cases} p \xrightarrow{*} u_1 \\ q \xrightarrow{*} u_1 \end{cases}$$

**Definition 4.4.2** *Un système de réécriture est dit* localement *confluent* **ssi**

$$\forall u_0, p, q \in \mathcal{A} \quad \textit{tels que} \quad \begin{cases} u_0 \rightarrow p \\ u_0 \rightarrow q \end{cases}$$

$$\exists u_1 \in \mathcal{A} \quad \textit{tel que} \quad \begin{cases} p \xrightarrow{*} u_1 \\ q \xrightarrow{*} u_1 \end{cases}$$

Cette propriété est plus faible que la confluence, elle suppose seulement que les calculs peuvent confluer après un seul pas de réécriture.

**Théorème 4.4.1** *Un système de réécriture noethérien et localement confluent est confluent.*

Preuve: Nous allons utiliser le lemme suivant : voir figure 4.1.

**Lemme 4.4.1** *Si le système $\mathcal{R}$ est confluent pour tout polynôme $v$ obtenu à partir de $u_0$ en une seule réécriture, alors il est confluent en $u_0$.*

Supposons qu'il existe $u_0$ pour lequel le système n'est pas confluent. Le lemme assure l'existence d'un polynôme $u_1$ (dérivant directement de $u_0$) pour lequel le système n'est pas confluent. On construit ainsi une chaîne infinie $u_0 \rightarrow u_1 \rightarrow \cdots$, ce qui contredit l'hypothèse de noethérianité. □

[4] L'ordre lexicographique convient pour l'étude d'un idéal à gauche

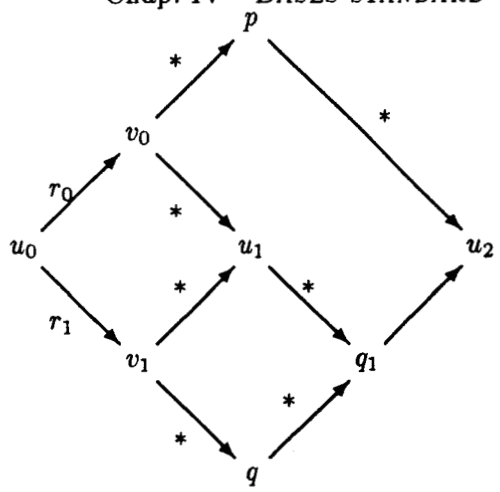



Figure 4.1 :

### 4.4.1 Les paires critiques

Intuitivement, une paire critique est la donnée de deux substitutions portant sur deux facteurs d'un même mot $w$, les facteurs se chevauchant ; la paire est alors *critique* au sens où la confluence locale sur le mot $w$ n'est pas assurée.

Dans cette section, on supposera que $\mathcal{W}$ est un monoïde (cette hypothèse n'ést pas utile pour les bases-standard d'espaces vectoriels).

**Definition 4.4.3** *On appelera* paire critique *la donnée de deux substitutions*

$$\begin{cases} uc & \xrightarrow{r_1} & p \\ cv & \xrightarrow{r_2} & q \end{cases}$$

*avec* $u, c, v \in \mathcal{W}$, $p, q \in \mathcal{A}$ *et* $c \neq \varepsilon$.

Le mot $w = ucv$ se réécrit de deux façons différentes

$$\begin{cases} w & \xrightarrow{r_1} & pv \\ w & \xrightarrow{r_2} & uq \end{cases}$$

sans que la confluence locale soit acquise.

Le polynôme $s = pv - uq$ est appelé un $S$.polynôme.

Remarque: on a la propriété de *confluence locale* à partir du mot $w$ **ssi** $s \xrightarrow{\mathcal{R}_G} 0$. [5]

#### 4.4.1.1 Réduction d'une paire critique

Contrairement à ce qui se passe dans la réécriture des arbres (algèbres de termes), la réduction d'une paire critique donnée est toujours possible.

On a $s \equiv_E 0$ ; considérons une forme normale $\bar{s}$ de $s$ dans $\mathcal{R}_G$. Lorsque $\bar{s} \neq 0$, on assure la

[5] On pourra constater sur les divers exemples que la notion de *paire critique* est en fait dépendante de la technique de filtrage.

confluence locale sur la paire critique en ajoutant à $G$ le polynôme $\frac{1}{lc(\bar{s})}\bar{s}$.

#### 4.4.1.2 Exemple

Soit $X = \{a, b, c\}$ muni de l'ordre lexicographique par longueur. On considère les règles:

$$\begin{cases} ab \longrightarrow 1 & (r_1) \\ bc \longrightarrow 2 & (r_2) \end{cases}$$

en adoptant comme filtrage la relation :

$u \preceq v$ **ssi** $u$ est un facteur de $v$

On obtient une paire critique en observant que le mot $w = abc$ peut s'écrire de 2 façons:

$w \xrightarrow{r_1} c$ et $w \xrightarrow{r_2} 2a$

On obtient par réduction la règle: $c \longrightarrow 2a$.

**Remarque:** On peut générer des paires critiques à partir d'une seule règle:

$aba \longrightarrow 1$

On obtient une paire critique en considérant le mot $ababa$:

$$\begin{cases} (aba)ba \longrightarrow ba \\ ab(aba) \longrightarrow ab \end{cases}$$

d'où la nouvelle règle $ba \longrightarrow ab$.

**Proposition 4.4.1** *Un système de réécriture localement confluent sur ses paires critiques est localement confluent.*

La preuve de cette proposition tient dans le fait que lorsque deux substitutions ne forment pas une paire critique, l'ordre dans lequel on les utilise est indifférent; le diagramme de la figure 4.2 page 65, commute.

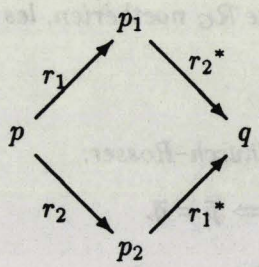



Figure 4.2 :

## 4.5 Algorithme de Knuth–Bendix

L'algorithme de Knuth-Bendix consiste à compléter un système de réécriture noethérien pour le rendre localement confluent sur ses paires critiques, donc confluent. La seule véritable difficulté est de prouver que cette complétion n'entraîne pas l'ajout d'un nombre infini de règles[6].

**Théorème 4.5.1** *Dans un système confluent noethérien, tout polynôme admet une forme normale (c'est-à-dire réduite) unique.*

La preuve est immédiate : la noethérianité implique l'existence d'une forme réduite obtenue par un nombre fini de substitutions. La confluence implique l'unicité de ces formes normales. L'ordre dans lequel on applique les règles pour obtenir la forme normale est donc indifférent.

### 4.5.1 La complétude

La relation entre polynômes $p \xrightarrow{*} q$ est réflexive et transitive. Sa clôture symétrique définit une relation d'équivalence que nous noterons $\equiv_{\mathcal{R}}$.

**Definition 4.5.1** *Un système $\mathcal{R}$ est dit complet pour la relation d'équivalence $\equiv_E$* **ssi** *les deux relations $\equiv_{\mathcal{R}}$ et $\equiv_E$ sont égales.*

Intuitivement, cela signifie que $\forall p, q \in \mathcal{A}$ tels que $p \equiv_E q$, $\exists p_0, p_1, p_2, \ldots p_n$, tels que:

$$p = p_0 \xrightarrow{*} p_1 \xleftarrow{*} p_2 \xrightarrow{*} \ldots \xleftarrow{*} p_n = q$$

### 4.5.2 Caractérisation d'une base–standard

On notera $\mathrm{lt}(G)$ l'ensemble des mots de tête[7] des polynômes de l'ensemble $G$.

Soient $U$ et $V$ deux ensembles de mots. On dira que $U$ filtre $V$ **ssi**

$$\forall y \in V, \quad \exists x \in U, \quad x \preceq y$$

On considère une congruence $\equiv_E$ associée à un espace vectoriel $E$, $G$ un ensemble de polynômes de $E$ et $\mathcal{R}_G$ un système de réécriture associé à $G$.

**Théorème 4.5.2** *Pour un système $\mathcal{R}_G$ noethérien, les propriétés suivantes sont équivalentes:*

1. *$\mathcal{R}_G$ est complet et confluent.*
2. *$\mathcal{R}_G$ vérifie la condition de Church-Rosser:*

$$\forall p, q \in \mathcal{A}, \quad p \equiv_E q \iff \overline{p} = \overline{q}.$$

3. $\forall p \in \mathcal{A}, \quad p \in E \iff p \xrightarrow{*} 0.$
4. *$G \subset E$ et* $\mathrm{lt}(G)$ filtre $\mathrm{lt}(E)$.

[6] Ceci se produit effectivement dans le calcul de la base–standard d'un idéal différentiel.

[7] J'espère que le lecteur n'attrapera pas la migraine !

*La forme normale d'un polynôme est alors le plus petit élément de sa classe d'équivalence.*

**Preuve:**

- $1 \Longrightarrow 2$ Soient $p$ et $q$ tels que

  $$\exists p_0, p_1, p_2, ...p_n, \quad \text{tels que } p = p_0 \xrightarrow{*} p_1 \xleftarrow{*} p_2 \xrightarrow{*} ... \xleftarrow{*} p_n = q$$

  Pour $n = 2$, la preuve est triviale (voir figure 4.3)

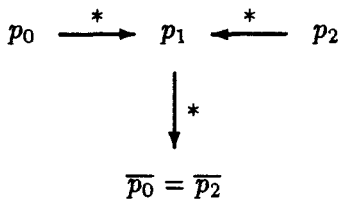



Figure 4.3 : cas $n = 2$

Pour $n > 2$, on peut remplacer la suite

$$p_0 \xrightarrow{*} p_1 \xleftarrow{*} p_2 \xrightarrow{*} ... \xleftarrow{*} p_n$$

par la suite

$$p_0 \xrightarrow{*} \bar{p}_2 \xleftarrow{*} p_4 \xrightarrow{*} ... \xleftarrow{*} p_n$$

ce qui diminue la longueur de la liste, puis on réitère le processus (voir figure 4.4).

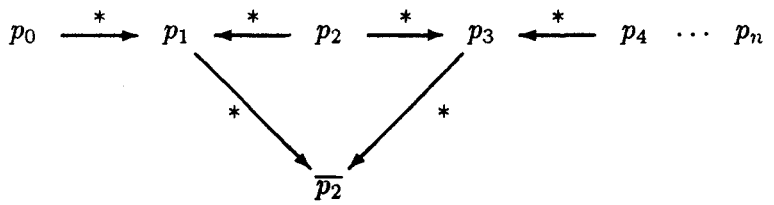



Figure 4.4 : cas $n > 2$

□

- $2 \Longrightarrow 1$ évident. □
- $2 \Longleftrightarrow 3$ et $3 \Longleftrightarrow 4$ sont faciles. □

### 4.5.3 Principe de l'algorithme de complétion

L'algorithme de Knuth-Bendix dans sa version naïve se résume à :

BaseStandard (G: ListeDePolynômes): ListeDePolynômes ==
Initialisation:
$R$: ListeDePolynômes := $G$
tq $\mathcal{R}_R$ comporte des paires critiques non réduites faire
• Réduire ces paires critiques et compléter $R$ avec les formes réduites des $S$.polynômes obtenus.
• Calculer les nouvelles paires critiques apparues.
ftq
*retourner*($R$)

#### 4.5.3.1 Preuve d'arrêt

La preuve ne peut pas se faire dans le cas général puisque, dans le cas des idéaux différentiels, la base standard peut être infinie et donc l'algorithme ne termine pas.

## 4.6 Normalisation d'un système confluent noethérien

### 4.6.1 Motivation

On souhaite représenter canoniquement les espaces vectoriels et les idéaux de polynômes par leur base–standard. La donnée d'un espace vectoriel $E$ (resp. d'un idéal $I$) revient à se donner une relation d'équivalence à savoir la congruence modulo $E$ (resp. modulo $I$). On va voir que l'unicité des bases qui reconnaissent la congruence peut être obtenue à condition :

1. d'éliminer les règles superflues,
2. de réduire les parties droites de règles (en utilisant les autres règles de la base).

**Definition 4.6.1** *Soit $\mathcal{R}$ un système de réécriture quelconque. Une règle $r \in \mathcal{R}$ de la forme $w \xrightarrow{r} q$ est dite* superflue **ssi** *il existe dans $\mathcal{R}$ une autre règle $w_1 \to q_1$ permettant de réduire le mot $w$ ie. si $w_1 \preceq w$.*

**Proposition 4.6.1** *Soit $\mathcal{R}$ un système de réécriture confluent et complet par rapport à une relation d'équivalence. Une règle superflue peut être supprimée sans perdre, ni la confluence, ni la complétude.*

**Definition 4.6.2** *Un système confluent et noethérien est dit* normalisé **ssi**

- *Il ne possède aucune règle superflue,*
- *Les parties droites de règle sont complètement réduites.*

La base-standard $G$ est dite **complètement réduite ssi** $\mathcal{R}_G$ est normalisé.

**Théorème 4.6.1** *Pour une même relation d'équivalence $\equiv_E$ et pour un ordre de réduction donné (ordre sur les mots), deux systèmes de réécriture normalisés comportent exactement les mêmes règles.*

**Preuve:** Soient $\mathcal{R}_1$ et $\mathcal{R}_2$ deux systèmes vérifiant les hypothèses du théorème.
D'aprés le théorême 4.5.2, $\mathcal{R}_1$ et $\mathcal{R}_2$ calculent les mêmes formes normales.
Montrons que toute règle $w \to q$ de $\mathcal{R}_1$ appartient aussi à $\mathcal{R}_2$.

| | | |
|---|---|---|
| On a: $N_2(w)$ | $= N_2(q)$ | car $w \equiv q$ |
| | $= N_1(q)$ | car $\mathcal{R}_1$ et $\mathcal{R}_2$ calculent les mêmes formes normales |
| | $= q$ | car $q$ est réduit dans $\mathcal{R}_1$ |

Donc $w \xrightarrow{\mathcal{R}_2} q$.
Supposons que la règle $w \to q$ ne figure pas dans $\mathcal{R}_2$; ce système contiendrait alors une autre règle $w_1 \to q$ telle que $w_1 \preceq w$.
De même $w_1 \equiv q \Longrightarrow N_1(w_1) = q$ et donc $\mathcal{R}_1$ contient une règle $w_2 \to q$ avec $w_2 \preceq w_1$.
On a donc dans $\mathcal{R}_1$ les deux règles $w \to q$ et $w_2 \to q$ avec $w_2 \preceq w$. La règle $w \to q$ est donc superflue d'où la contradiction. □

## 4.7 Les espaces vectoriels de polynômes

### 4.7.1 Motivation

Nous allons exposer une méthode permettant de traiter la plupart des problèmes relevant de l'algèbre linéaire. Elle suppose seulement que l'on calcule les bases–standard pour des sous-espaces vectoriels finiment engendrés d'un espace vectoriel ayant une base complètement ordonnée mais pas forcément finie. C'est le cas des polynômes construits sur un alphabet quelconque (pas forcément fini) où la base canonique est le monoïde libre. C'est également le cas des polynômes de Lie avec les bases classiques de Hall ou de Lyndon. C'est enfin le cas lorsqu'on étudie les dépendances linéaires d'un ensemble de séries rationnelles.

Les exemples que nous donnerons sont pris dans l'algèbre des polynômes non commutatifs, mais l'implantation en SCRATCHPAD est générique et accepte en paramètre n'importe quel espace vectoriel ayant une base complètement ordonnée.

L'intérêt de cette présentation est uniquement pratique car la théorie en est très élémentaire ; il s'agit plutôt de rompre avec un vieux réflexe en calcul formel où *algèbre linéaire* rime avec *calcul matriciel.*

**Exemple**

Soit à calculer le rang de Lie $d$ d'un polynôme $g \in K\langle X\rangle$. On a par définition:

$$d = rang(g \triangleright \mathcal{L}ie\langle X\rangle).$$

Cela revient à calculer le rang de l'application linéaire:

$$p \in \mathcal{L}ie\langle X\rangle \longrightarrow g \triangleright p \in K\langle X\rangle.$$

Cette application linéaire n'est pas donnée par une matrice et de fait, le calcul de cette matrice est une perte de temps [8].

Il est plus astucieux de calculer une base-standard de $g \triangleright \mathcal{L}ie\langle X\rangle$.

[8] Notre expérience prouve que cette méthode n'est pas la plus efficace, surtout si l'on travaille avec des matrices dont les lignes et les colonnes sont indicées avec des entiers: il faudrait pouvoir indicer par des mots, ce qui n'est généralement pas le cas.

### 4.7.2 Calcul d'une base–standard d'espace vectoriel

Soit $E$ un espace vectoriel engendré par un nombre fini de polynômes.
La technique générale s'applique en précisant que

- les substitutions se font par mot entier i.e

$$u \preceq v \quad \textbf{ssi} \quad u = v$$

- tout ordre total sur les mots est admissible.

Il est facile de vérifier qu'un ensemble $G$ de polynômes engendre l'espace $E$ **ssi** le système de réécriture $\mathcal{R}_G$ est *complet* pour la congruence par rapport à $E$.

#### 4.7.2.1 Réduction des paires critiques

Deux règles $w_1 \xrightarrow{r_1} q_1$ et $w_2 \xrightarrow{r_2} q_2$ forment une paire critique **ssi** $w_1 = w_2$.

La réduction engendre un S.polynôme $q_1 - q_2$ qui est non nul **ssi** $q_1 \neq q_2$. Dans tous les cas, on peut supprimer immédiatement l'une des 2 règles $r_1$ ou $r_2$. Lorsque $q_1 = q_2$, une règle disparait.

**Le nombre total de règles ne peut pas augmenter au cours de l'algorithme.**

La base $G$ est standard **ssi** $\mathrm{lt}(G) = \mathrm{lt}(E)$.

**Lemme 4.7.1** *Les polynômes d'une base complètement réduite sont linéairement indépendants.*

**Preuve** Puisque $\mathcal{R}$ ne comporte pas de règles superflues, les mots de tête de $G$ sont tous différents ; ceci assure l'indépendance linéaire des polynômes de $G$.

#### 4.7.2.2 Complexité du calcul

On vérifiera facilement que ce calcul est équivalent à la triangularisation d'une matrice par la méthode du pivot de Gauss. La complexité est donc la même, à ceci près qu'on fait l'économie du calcul de la matrice.

#### 4.7.2.3 Exemple

Soit $X = \{x, y\}$ un alphabet. Et soit $E$ le sous–espace vectoriel de $\mathbb{Q}\langle X\rangle$ engendré par les polynômes

$$\{xy - 1,\ x^2 + 2xy + y^2,\ y^2 + 1\}$$

Le système $\mathcal{R}$ initial est formé des règles:

$$\begin{cases} xy & \to 1 & (1) \\ y^2 & \to -2xy - x^2 & (2) \\ y^2 & \to -1 & (3) \end{cases}$$

On obtient finalement le système normalisé:

$$\begin{cases} x^2 & \to -1 \\ xy & \to 1 \\ y^2 & \to -1 \end{cases}$$

### 4.7.3 Noyau et image d'un morphisme d'espace vectoriel

Soit $f : E \longrightarrow F$ une application linéaire où l'espace vectoriel $E$ est donné par un ensemble fini de générateurs $G_E$. Il s'agit de calculer la base–standard des deux espaces vectoriels $Ker(f)$ et $Im(f)$.

#### 4.7.3.1 Image

On calcule une base–standard de l'ensemble $f(G_E)$.

#### 4.7.3.2 Noyau

On considère l'espace vectoriel :

$$H = \{(x, y) \in E \times F \mid y = f(x)\}$$

On calcule une base–standard de cet espace vectoriel engendré par un nombre fini de couples $\{(x, f(x)) \mid x \in G_E\}$. Le calcul se fait en réduisant les paires critiques figurant dans la deuxième composante. Il apparait, au cours de l'élimination, des couples de la forme $(x_i, 0)_{1 \leq i \leq k}$. La base–standard complètement réduite de la famille $\{x_i\}_{1 \leq i \leq k}$ est une base de $ker(f)$.

### 4.7.4 Somme et intersection de deux espaces vectoriels

Soient $E$ et $F$ deux espaces vectoriels donnés par leur base–standard $G_E$ et $G_F$. Le but est de calculer la base standard de $E + F$ et de $E \cap F$.

#### 4.7.4.1 Somme

$E+F$ admet $G_E \cup G_F$ comme système générateur ; il suffit donc de calculer la base–standard associée.

#### 4.7.4.2 Intersection

On considère l'application linéaire :

$$p \in F \longrightarrow N_E(p)$$

où $N_E(p)$ désigne la forme normale de $p$ dans la base $G_E$. Le noyau de cette application linéaire est $E \cap F$.

### 4.7.5 Implantation en Scratchpad

Afin d'implanter le calcul en toute généralité, nous allons définir la catégorie des modules libres sur une base totalement ordonnée :

```
)abbrev category FMCAT FreeModuleCat

FreeModuleCat(R: Ring, Basis: OrderedSet):Category == Module(R) with
   leadingTerm : $ -> Basis
   leadingCoef : $ -> R
   reductum    : $ -> $
   monom       : (Basis, R) -> $
```

Le calcul de base-standard peut s'effectuer sur n'importe quel **FreeModule** à coefficients dans un corps. En fait, il suffirait que l'anneau des coefficients soit commutatif et intègre. L'algorithme de Bareiss permet de réaliser l'élimination Gaussienne sans passer dans le corps des fractions et en limitant la croissance de la taille des coefficients.

Nous avons vu en 4.7.3.2 qu'il est utile de travailler sur des couples de polynômes et de mener le travail d'élimination sur une des composantes. Le constructeur de domaine **VSBASIS1** concrétise cette idée [9].

```
)abbrev domain VSBASIS1 VectorSpaceStandardBasis1

VectorSpaceStandardBasis1(Basis, K, FM, VS): Public == Private where
  Basis: OrderedSet
  K : Field
  VS : Module(K)            -- Espace vectoriel
  FM : FreeModuleCat(K, Basis)
  COUPLE  ==> DirectSumModule(K, FM, VS)
  COUPLES ==> List COUPLE
  NNI     ==> NonNegativeInteger

  Public == Set with
    "*" :  $ -> NNI
    coerce : $ -> COUPLES
    CharacteristicSet: $ -> List Basis
    elt : ($,I) -> COUPLE
    NormalForm: (COUPLE,$) -> COUPLE
    stdbasis: COUPLES -> $

  Private == add
    import(COUPLE,Basis,FM)
```

[9] L'élimination se fait sur la première composante qui appartient à la catégorie des **FreeModule** tandis que l'autre composante est un module quelconque.

```
  -- representation
    Rep := COUPLES

  -- locales
    AvecPairesCritiquesReduites: COUPLES -> COUPLES
    AvecPartiesDroitesReduites : COUPLES -> COUPLES
    ....
```

Le constructeur de domaine VSBASIS implante la représentation canonique des espaces vectoriels de dimension finie par leur bases–standard:

```
)abbrev domain VSBASIS VectorSpaceStandardBasis

VectorSpaceStandardBasis(Basis, K, VS): Public == Private where
  Basis: OrderedSet
  K : Field
  VS : FreeModuleCat(K, Basis)              -- Espace vectoriel
  LVS ==> List VS
  NNI ==> NonNegativeInteger

  Public == Set with
    "#" :  $ -> NNI
    "+" : ($,$) -> $
    0   : constant -> $
    coerce : $ -> LVS
    CharacteristicSet: $ -> List Basis
    elt : ($,I) -> VS
    intersection: ($,$) -> $
    NormalForm: (VS,$) -> VS
    stdbasis: LVS -> $

  Private == add
  -- representation
    Rep := LVS
  ....
```

## 4.8 Les idéaux de polynômes non commutatifs

### 4.8.1 Motivation

Les idéaux de polynômes non commutatifs interviennent dans l'étude des séries rationnelles initiée par M.P. Schützenberger en 1960. On doit à C. Reuteunauer [30] la notion d'algèbre et d'idéal syntaxique[10]. Nous avons en tête une représentation de ces séries qui utiliserait les techniques de base–standard en remplacement de la traditionnelle représentation matricielle.

[10] C'est le pendant, pour les séries, de la notion de monoïde syntaxique en théorie des langages rationnels.

## 4.8.2 Calcul de la base–standard d'un idéal à droite

### 4.8.2.1 Filtrage

Les substitutions se font en partie gauche des mots i.e.

$u \preceq v$ **ssi** $u$ est un facteur gauche de $v$.

On vérifiera sans peine qu'un ensemble $G$ de polynômes engendre un idéal à droite $I$ **ssi** le système de réécriture $\mathcal{R}_G$ est complet pour la congruence par rapport à $I$.

### 4.8.2.2 Ordre admissible

On a vu en 4.3.1 que l'ordre lexicographique n'est pas admissible ; nous travaillerons avec l'ordre lexicographique par longueur.

### 4.8.2.3 Paires critiques

Les règles $u_1 \xrightarrow{r_1} q_1$ et $u_2 \xrightarrow{r_2} q_2$ correspondant aux polynômes générateurs $g_1 = u_1 - q_1$ et $g_2 = u_2 - q_2$ forment une paire critique **ssi**

$u_1$ est facteur gauche de $u_2$

Posons $u_2 = u_1 f$. La réduction de cette paire critique conduit en général à ajouter la règle correspondant au $S$.polynôme $s = q_2 - q_1 f$.

De fait, on peut mais supprimer immédiatement la règle $u_2 \xrightarrow{r_2} q_2$. Ceci est justifié par le fait que les paires $\{g_1, g_2\}$ et $\{g_1, s\}$ engendrent le même idéal à droite.

Exemple

$$\begin{cases} a^2b & \longrightarrow 2a^2 + 1 & (r_1) \\ a^2 & \longrightarrow a & (r_2) \end{cases}$$

est remplacé par

$$\begin{cases} ab & \longrightarrow 2a^2 + 1 & (r_1') \\ a^2 & \longrightarrow a & (r_2) \end{cases}$$

**Le nombre de règles ne peut pas augmenter au cours de l'algorithme.**

### 4.8.2.4 Preuve d'arrêt

Chaque réduction de paire critique conduit à remplacer un polynôme de $G$ par un polynôme strictement inférieur ; donc le processus s'arrête. □

A la fin, on obtient un système qui n'a plus de paires critiques.

#### 4.8.2.5 Codes préfixes

**Definition 4.8.1** *Soit $C$ un ensemble de mots. Le code $C$ est dit* préfixe **ssi**

$$\forall u, f \in X^*, \quad u \in C \text{ et } uf \in C \Longrightarrow f = \varepsilon$$

A tout code préfixe $C$, on associe une partie préfixielle $P$ définie par:

$$P = \{u \in X^* \mid \exists f \in X^*; uf \in C\}$$

On a: $P = X^* - CX^*$.

**Definition 4.8.2** *Le code préfixe est dit* complet **ssi** $X^* = C^*P$.

**Exemple :**

$$C = \{aab, aaa, ab, ba, bb\}$$

**Remarque:** Les mots d'un code préfixe complet fini $C$ constituent les feuilles d'un arbre binaire dont les noeuds internes constituent la partie préfixielle (voir figure 4.5).

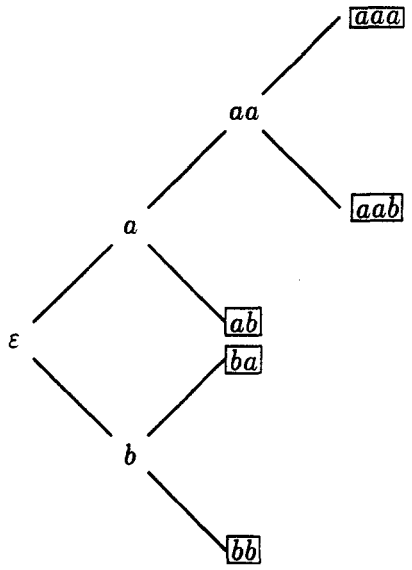



Figure 4.5 :

**Proposition 4.8.1** *Si $G$ est une base standard sans règle superflue, les têtes de règle forment un code préfixe.*

**Preuve:** immédiate en utilisant les définitions.□

### 4.8.3 Idéaux de codimension finie

**Definition 4.8.3** *Un idéal $I \subset K\langle X\rangle$ est dit de* codimension finie **ssi** *le quotient $K\langle X\rangle/I$ est un espace vectoriel de dimension finie.*

**Exemple:** Considérons deux matrices carrées $A$ et $B$ de même dimension. L'algèbre $K\langle A, B\rangle$ est donc un espace vectoriel de dimension finie. Cette algèbre peut être décrite par l'idéal $I$ de codimension finie :

$$I = \{p \in K\langle x, y\rangle \mid p(A, B) = 0\}$$

Nous verrons que $I$ est finiment engendré en tant qu'idéal à droite.

**Théorème 4.8.1** *Pour un idéal à droite $I$ dont la base standard complètement réduite est $G$, les propriétés suivantes sont équivalentes:*

1. *$K\langle X\rangle/I$ est un espace vectoriel de dimension finie.*
2. *tete$(G)$ est un code préfixe* complet *et fini.*
3. *Les formes normales sont de degré borné i.e*

$$\exists l \in \mathbb{N} \text{ tel que } \forall p \in K\langle X\rangle, \quad degre(\bar{p}) \leq l.$$

**Preuve:**
la démonstration est facile si l'on remarque que l'espace vectoriel $K\langle X\rangle/I$ est canoniquement représenté par les formes normales de $K\langle X\rangle$. □

**Théorème 4.8.2 (Levin 1969)** *Tout idéal à droite de codimension finie $c$ est finiment engendré. La dimension $g$ de sa base standard réduite est donnée par la formule:*

$$g = c(n-1) + 1$$

*où $n$ désigne le cardinal de l'alphabet $X$.*

**Preuve:** cette formule découle de la relation qui lie le nombre de feuilles $g$ et le nombre de noeuds internes $c$ d'un arbre $n$–aire quelconque. □

### 4.8.4 Implantation en SCRATCHPAD

Cet algorithme est en cours d'implantation. Il semble que la meilleure structure de données pour représenter la base ne soit pas la liste mais la structure d'arbre ; le calcul de forme normale s'en trouve accéléré.

Si l'on adopte la représentation récursive des polynômes (arborescente), l'algorithme qui en découle est joliment récursif.

## 4.9 Application: représentation des séries rationnelles

### 4.9.1 Rationalité et reconnaissabilité

Soit $X$ un alphabet et $K$ un corps; on note

$$S^* = \sum_{n=0}^{\infty} S^n$$

cette somme ayant un sens si $< S \mid \varepsilon > = 0$.

**Definition 4.9.1** *L'ensemble des séries rationnelles est le plus petit ensemble contenant les lettres de $X$ et les constantes de $R$ et fermé par les opérations rationnelles "$+$", "$\times$" et "$*$".*

On note

$$S \triangleright K\langle X\rangle = \{S \triangleright p \mid p \in K\langle X\rangle\}$$
$$K\langle X\rangle \triangleleft S = \{p \triangleleft S \mid p \in K\langle X\rangle\}$$

**Definition 4.9.2** *Une série $S$ est dite* reconnaissable **ssi** *elle vérifie l'une des propriétés équivalentes suivantes :*

1. *$S \triangleright K\langle X\rangle$ est un espace vectoriel de dimension finie.*
2. *$K\langle X\rangle \triangleleft S$ est un espace vectoriel de dimension finie.*

Les deux espaces vectoriels ont alors même dimension ; ce rang est appelé *rang de Hankel* de la série $S$.

**Théorème 4.9.1 (Schützenberger)** *Les deux notions de rationalité et de reconnaissabilité sont équivalentes.*

### 4.9.2 Annulateurs d'un ensemble de séries

Soit $E \subset K\langle\langle X\rangle\rangle$ un ensemble de séries. L'annulateur à droite de $E$ est défini par

$$Ann_d(E) = \{p \in K\langle X\rangle \mid \forall S \in E, S \triangleright p = 0\}$$

On définit de même l'annulateur à gauche de $E$

$$Ann_g(E) = \{p \in K\langle X\rangle \mid \forall S \in E, p \triangleleft S = 0\}$$

**Proposition 4.9.1** *L'annulateur à droite (resp à gauche) d'une série rationnelle $S$ est un idéal à droite (resp à gauche) de codimension finie.*

On a en effet : $S \triangleright K\langle X\rangle \simeq K\langle X\rangle / Ann_d(S)$. □

### 4.9.3 Idéal et algèbre syntaxiques

L'ensemble orthogonal d'une série $S$ est défini par

$$S^{\perp} = \{p \in K\langle X\rangle \mid < S \mid p >= 0\}$$

L'idéal syntaxique $\mathcal{J}(S)$ d'une série $S$ est défini par l'une des propriétés équivalentes suivantes :

1. $\mathcal{J}(S) = Ann_d(S \triangleright K\langle X\rangle)$
2. $\mathcal{J}(S) = Ann_g(K\langle X\rangle \triangleleft S)$
3. $\mathcal{J}(S) = \{p \in K\langle X\rangle \mid \forall u, v \in X^*, < S \mid upv >= 0\}$
4. $\mathcal{J}(S)$ est le plus grand idéal bilatère contenu dans $S^{\perp}$.

**Definition 4.9.3** *L'algèbre syntaxique d'une série $S$ est l'algèbre quotient*
$Synt(S) = K\langle X\rangle/\mathcal{J}(S)$.

**Théorème 4.9.2 (C.Reutenauer)** *Une série est rationnelle* **ssi** *son algèbre syntaxique est un espace vectoriel de dimension finie.*

**Preuve:** Voir [1]. □

Soit $(\lambda, \mu, \gamma)$ une représentation linéaire de $S$. On a alors par définition
$\forall w \in X^*, \quad < S \mid w > \quad = \quad \lambda\mu_w\gamma$

**Théorème 4.9.3 (C.Reutenauer)** *Si la représentation linéaire $(\lambda, \mu, \gamma)$ est minimale, l'algèbre syntaxique est isomorphe à l'algèbre $\{\mu_p \mid p \in K\langle X\rangle\}$.*

**Preuve:** Voir [1]. □

### 4.9.4 Représentation canonique d'une série rationnelle

Soit à représenter une série rationnelle $S$ ; considérons son annulateur à droite $\mathcal{A}n_d(S)$ qu'on peut définir par sa base standard (complètement réduite). Recherchons l'information supplémentaire qui nous permettrait de calculer le coefficient de n'importe quel mot $w$ dans $S$.

Soit $\bar{w}$ la forme normale de $w$ dans la base standard de $\mathcal{A}n_d(S)$.
On a : $w - \bar{w} \in \mathcal{A}n_d(S)$ d'où $S \triangleright w = S \triangleright \bar{w}$ et par suite $< S|w > = < S|\bar{w} >$.

Soit $S_{\mathcal{P}}$ le polynôme correspondant à la série $S$ tronquée sur la partie préfixielle $\mathcal{P}$ de la base standard ie.

$$S_{\mathcal{P}} \quad = \quad \sum_{w \in \mathcal{P}} < S|w > w$$

On a alors : $< S|w > = < S|\bar{w} > = < S_{\mathcal{P}}|\bar{w} >$.

Ainsi la série $S$ est la donnée :

1. de la base standard de son annulateur à droite.
2. d'un polynôme $S_{\mathcal{P}}$ correspondant aux premiers termes de $S$.

#### 4.9.4.1 Exemple

On considère l'alphabet $X = \{x_0, x_1\}$.
Soit $S$ la série $S = \sum_{w \in X^*} \varepsilon(w)w$ où $\varepsilon(w)$ désigne le nombre entier codé par le mot $w$ interprété comme une suite de bits 0 ou 1. Ainsi

$$S \quad = \quad x_1 + x_0x_1 + 2x_1x_0 + 3x_1x_1 + x_0^2x_1 + 2x_0x_1x_0 + 3x_0x_1^2 + \ldots$$

Montrons que $S$ est rationnelle et pour celà calculons ses résiduels ; on a :

$$\begin{aligned} S \triangleright x_0 &= \sum_{w\in X^*} \varepsilon(x_0 w)w &= S \\ S \triangleright x_1 &= \sum_{w\in X^*} \varepsilon(x_1 w)w &= \sum_{w\in X^*} \Big(2^{|w|} + \varepsilon(w)\Big)w \\ &= S + T \end{aligned}$$

en posant $T = \sum_{w\in X^*} 2^{|w|}w$.

Le calcul des résiduels de $T$ donne:

$$T \triangleright x_0 = \sum_{w\in X^*} 2^{|x_0 w|}w = 2T$$

$$T \triangleright x_1 = \sum_{w\in X^*} 2^{|x_1 w|}w = 2T$$

On a : $T = S \triangleright x_1 - S$.

La relation $T \triangleright x_0 = 2T$ donne $S \triangleright x_1 x_0 - S \triangleright x_0 = 2S \triangleright x_1 - 2S$

d'où la règle : $x_1 x_0 \longrightarrow x_0 + 2x_1 - 2$.

De la relation $T \triangleright x_1 = 2T$, on déduit de même: $x_1^2 \longrightarrow 3x_1 - 2$.

On obtient finalement le système de réécriture complètement normalisé:

$$(\mathcal{R}_G) \quad \begin{cases} x_0 \longrightarrow 1 \\ x_1 x_0 \longrightarrow 2x_1 - 1 \\ x_1^2 \longrightarrow 3x_1 - 2 \end{cases}$$

La série est donc canoniquement représentée par la donnée de la base–standard $G$ et du polynôme $S_\mathcal{P} = x_1$. Cette représentation correspond à l'automate de la figure 4.7 et à l'arbre de la figure 4.7.

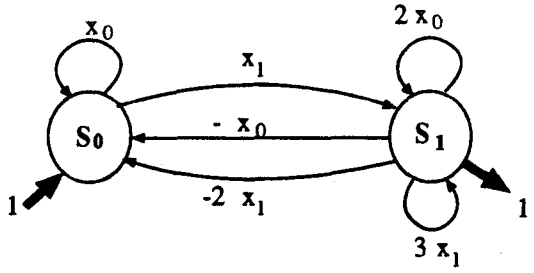



Figure 4.6 : Automate de S

### 4.9.5 Calcul du rang de Lie d'une série rationnelle

**Definition 4.9.4** *Le rang de Lie d d'une série quelconque S correspond à la dimension de l'espace vectoriel* $S \triangleright \mathcal{L}ie\langle X\rangle$ *ie.*

$$d = \text{rang}\{S \triangleright p \,|\, p \in \mathcal{L}ie\langle X\rangle\}$$

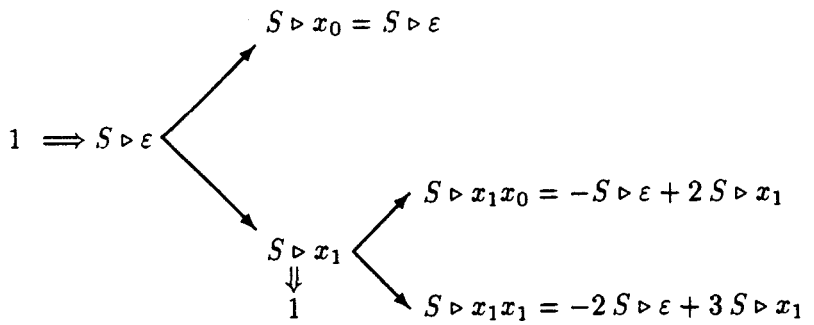



Figure 4.7 : Arbre

**Remarque:** *Le rang de Lie est inférieur ou égal au rang de Hankel*

On a: $S \triangleright \mathcal{L}ie_K\langle X\rangle \subseteq S \triangleright K\langle X\rangle$. □

Par définition, le rang de Hankel d'une série rationnelle est fini et donc son rang de Lie l'est également. Nous allons montrer que le calcul du rang de Lie d'une série rationnelle est facilité si l'on dispose d'une base standard de son idéal syntaxique et que celà évite de calculer des produits matriciels.

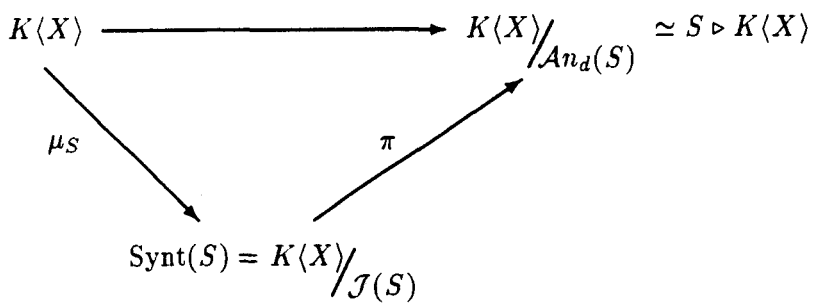



Figure 4.8 : Calcul du rang de Lie

On s'intéresse au rang de l'espace vectoriel : (voir figure 4.8)

$$S \triangleright \mathcal{L}ie\langle X\rangle \simeq \pi \circ \mu_S(\mathcal{L}ie\langle X\rangle)$$

$S$ étant rationnelle, l'algèbre syntaxique $\mathcal{S}ynt(S) = \mu_S(K\langle X\rangle)$ est un espace de dimension finie ; il en est donc de même pour $\mu_S(\mathcal{L}ie\langle X\rangle)$.

Le fait que l'idéal syntaxique $\mathcal{J}(S)$ soit bilatère ($\mu_S$ est un morphisme d'anneau) permet de calculer $\mu_S(\mathcal{L}ie\langle X\rangle)$ car on a :

$$\forall p, q \in \mathcal{L}ie\langle X\rangle, \qquad \mu_S[p,q] = [\mu_S(p), \mu_S(q)]$$

$\mu_S(\mathcal{L}ie\langle X\rangle)$ est donc une algèbre de Lie (de dimension finie) engendrée par $\mu_S(X)$.

#### 4.9.5.1 Algorithme

L'algorithme comporte deux parties:

1. Calcul de $E = \mu_S(\mathcal{L}ie\langle X\rangle)$
2. Calcul de $\pi(E)$

Le morphisme $\mu_S$ peut être assimilé à un calcul de forme normale $p \longrightarrow N_S(p)$ des polynômes $p \in \mathcal{L}ie\langle X\rangle$ dans le système de réécriture associé à une base standard de $\mathcal{J}(S)$.

$\mu_S(\mathcal{L}ie\langle X\rangle)$ peut être vu comme le plus petit espace vectoriel $E$ contenant $N_S(X)$ et stable pour le crochet :

$$(p,q) \in E \times E \longrightarrow N_S(pq - qp)$$

Il est donc facilement calculable par la méthode du point fixe appliquée aux espaces vectoriels.

Le morphisme $\pi$ peut être assimilé à un calcul de forme normale dans le système de réécriture associé à une base standard de $\mathcal{A}n_d(S)$.

# 5

# Algèbres de mélange

## 5.1 Motivation

Il s'agit de présenter un certain nombre de résultats théoriques permettant de comprendre l'algorithme de la réalisation minimale analytique.

La première partie traite de la base duale d'une base de Poincaré-Birkoff-Witt. Si cette dernière est construite à partir de produits décroissants de polynômes de Lyndon, la base duale est polynomiale et nous donnons un algorithme récursif de C.Reutenauer [21] pour la construire. Qu'on ne s'y trompe pas, cette bonne propriété est due au fait que les polynômes de Lyndon sont homogènes et ne se généralise pas à une base quelconque de l'algèbre de Lie libre $\mathcal{L}ie\langle X\rangle$.

La deuxième partie traite des sous-algèbres de mélange dont l'annulateur est une sous-algèbre de $\mathcal{L}ie\langle X\rangle$ de codimension finie. C.Reutenauer a démontré dans [30] que ces algèbres de mélange admettent une "base" finie. Malheureusement, l'examen de la preuve de ce théorème ne fournit pas un algorithme effectif de décomposition d'un polynôme dans la base, même lorsque celle-ci est formée de polynômes ; la décomposition est en effet obtenue en inversant un système linéaire triangulaire comportant une infinité d'équations.

## 5.2 Rappels sur le produit de mélange

Le produit de mélange a été défini à la page 27.

**Théorème 5.2.1** *Les résiduels par les polynômes de Lie sont des dérivations pour le produit de mélange i.e.*

$$\forall P \in \mathcal{L}ie\langle X\rangle,\ \forall S,T \in K\langle\langle X\rangle\rangle, \qquad (S \sqcup T) \triangleright P = (S \triangleright P) \sqcup T + S \sqcup (T \triangleright P)$$

**Preuve:** on sait que le calcul du résiduel par une lettre est une dérivation pour le shuffle (voir la section 2.6) ; ce résultat s'étend à l'algèbre de Lie des dérivations. □

Nous allons maintenant rappeler un formalisme utilisant le produit tensoriel et qui, en reformulant ce théorème, permet de raccourcir certaines preuves.

## 5.2.1 Produit de mélange et produit tensoriel

### 5.2.1.1 Coproduit d'ordre $n$

On note $K\langle X\rangle^{\otimes n} = K\langle X\rangle \otimes K\langle X\rangle \cdots \otimes K\langle X\rangle$ ($n$ facteurs).

**Definition 5.2.1** *Le coproduit $c_n : K\langle X\rangle \longrightarrow K\langle X\rangle^{\otimes n}$ est un morphisme d'algèbres, défini pour toute lettre $x \in X$ par*

$$c_n(x) = \sum_{i+1+j=n} 1^{\otimes i} \otimes x \otimes 1^{\otimes j}$$

**Exemple:**

$$\begin{aligned} c_2(a) &= a \otimes 1 + 1 \otimes a \\ c_2(b) &= b \otimes 1 + 1 \otimes b \\ c_2(ab) &= (a \otimes 1 + 1 \otimes a)(b \otimes 1 + 1 \otimes b) \\ &= ab \otimes 1 + a \otimes b + b \otimes a + 1 \otimes ab \end{aligned}$$

**Théorème 5.2.2 ([29, 14])** *Les polynômes de Lie sont des éléments primitifs i.e.*

$$\forall p \in \mathcal{L}ie\langle X\rangle, \quad c_n(p) = \sum_{i+1+j=n} 1^{\otimes i} \otimes p \otimes 1^{\otimes j}$$

**Preuve:** l'ensemble des éléments primitifs contient les lettres et est fermé par crochet de Lie. □

### 5.2.1.2 Nouvelle définition du produit de mélange

La dualité entre séries et polynômes est étendue au produit tensoriel par

$$< S_1 \otimes S_2 \otimes \cdots \otimes S_n | P_1 \otimes P_2 \otimes \cdots \otimes P_n > \;=\; < S_1|P_1 >< S_2|P_2 > \cdots < S_n|P_n >$$

**Definition 5.2.2** *Le produit de mélange de n séries est défini par la dualité :*

$$\forall P \in K\langle X\rangle, \qquad < S_1 \sqcup S_2 \cdots \sqcup S_n | P > \;=\; < S_1 \otimes S_2 \cdots \otimes S_n | c_n(P) >$$

On considère les produits scalaires de la forme

$$< S_1 \sqcup S_2 \cdots \sqcup S_n | P_1 P_2 \cdots P_k >$$

où $S_1, S_2 \ldots, S_n$ sont des séries propres et $P_1, P_2 \ldots, P_k$ sont des polynômes de Lie.

**Proposition 5.2.1** *Si $n > k$ alors*

$$< S_1 \sqcup S_2 \cdots \sqcup S_n | P_1 P_2 \cdots P_k > \;=\; 0$$

**Preuve:** le lecteur pourra se convaincre en effectuant le calcul pour $n = 2$ et $k = 1$.
Voir [30]. □

**Proposition 5.2.2**

$$< S_1 \sqcup\!\sqcup S_2 \cdots \sqcup\!\sqcup S_n | P_1 P_2 \cdots P_n > \;=\; \sum_{\sigma \in \mathcal{S}_n} < S_1 | P_{\sigma(1)} >< S_2 | P_{\sigma(2)} > \cdots < S_n | P_{\sigma(n)} >$$

*la somme étant calculée pour toutes les permutations $\sigma$ du groupe symétrique $\mathcal{S}_n$.*

**Preuve:** le lecteur pourra se convaincre en effectuant le calcul pour $n = 2$ :

$$< S_1 \sqcup\!\sqcup S_2 | P_1 P_2 > \;=\; < S_1 | P_1 >< S_2 | P_2 > + < S_1 | P_2 >< S_2 | P_1 >$$

Voir [30]. □

## 5.3 Base duale de la base $PBWL$

Rappelons que les éléments de la base $PBWL$ sont de la forme (voir 3.5.2 page 52) :

$$Q_w = [l_1]^{\alpha_1}[l_2]^{\alpha_2} \ldots [l_k]^{\alpha_k} \quad \text{avec } l_i \in \mathcal{L}yndon\langle X \rangle \text{ et } l_1 > l_2 > \cdots > l_k$$

et constituent une base de $K\langle X \rangle$ .

Lorsque $l$ désigne un mot de Lyndon, $Q_l = [l]$ désigne le polynôme de Lie associé à $l$.

Une série $S$ est une forme linéaire définie sur $K\langle X \rangle$ :

$$P \quad \xrightarrow{S} \quad < S | P >$$

On peut définir arbitrairement les nombres $< S|P >$ pour des polynômes $P$ formant une base de $K\langle X \rangle$, ici la base $PBWL$.

**Definition 5.3.1** *La base duale $\{S_w \in K\langle\langle X \rangle\rangle\}$ est définie par la relation d'orthogonalité :*

$$< S_w | Q_{w_1} > \;=\; \delta_w^{w_1} \quad \textit{avec } w, w_1 \in X^*.$$

**Proposition 5.3.1** *Les séries $S_w$ sont des polynômes homogènes de degré $|w|$.*

**Preuve:** il suffit de prouver que:

$$< S_w | w_1 >= 0 \quad \text{pour } |w| \neq |w_1|$$

Or ceci découle de la proposition 3.5.3 qui implique que $w_1$ se décompose dans $PBWL$ en composantes homogènes de degré $|w_1|$. □

Cette situation particulièrement agréable tient au fait que les éléments de la base de Lyndon sont des polynômes de Lie homogènes. La démonstration ne tient pas pour la base duale d'une base de Poincaré–Birkoff–Witt quelconque.

### 5.3.1 Algorithme de construction de la base duale

Les polynômes $S_w$ sont définis récursivement par les formules:

1. $S_\epsilon = 1$

2. $S_{xu} = xS_u$ si $x$ désigne la première lettre du mot de Lyndon $xu$.

3. $S_w = \dfrac{S_{l_1}^{\alpha_1} \sqcup S_{l_2}^{\alpha_2} \ldots \sqcup S_{l_k}^{\alpha_k}}{\alpha_1! \alpha_2! \ldots \alpha_k!}$
en supposant que $l_1^{\alpha_1} l_2^{\alpha_2} \ldots l_k^{\alpha_k}$ désigne la factorisation décroissante de $w \in X^*$.

**Preuve:** Voir G.Melançon et C.Reutenauer [21]. □

**Proposition 5.3.2** *Les polynômes $S_l$ où $l$ décrit l'ensemble des mots de Lyndon forment une base de l'algèbre de mélange. On a la décomposition :*

$$\forall T \in K\langle\langle X \rangle\rangle, \quad T = \sum_{w \in X^*} < T | Q_w > S_w$$

**Preuve:** Voir G.Melançon et C.Reutenauer [21]. □

Lorsque $T$ est un polynôme, cette somme est finie.

### 5.3.2 Factorisation de la série double

La série double $D$ est un élément de $K\langle\langle X \otimes X \rangle\rangle$ définie par la formule:

$$D = \sum_{w \in X^*} w \otimes w$$

**Lemme 5.3.1**

$$\sum_{w \in X^*} w \otimes w = \sum_{w \in X^*} S_w \otimes Q_w$$

**Preuve** Voir [12]. □

**Proposition 5.3.3**

$$\sum_{w \in X^*} w \otimes w = \prod_{l \in \mathcal{L}yndon\langle X \rangle} e^{S_l \otimes Q_l}$$

*Ce produit infini étant ordonné par ordre décroissant sur les mots de Lyndon.*

**Preuve** Voir [12]. □

## 5.4 Sous-algèbres définies par leur annulateur

Soit $\mathcal{A}$ une sous-algèbre de de Lie de $\mathcal{L}ie\langle X \rangle$.
On considère l'ensemble $\mathcal{V}(\mathcal{A})$ formé des séries annulées par $\mathcal{A}$ i.e.

$$\mathcal{V}(\mathcal{A}) = \{s \in K\langle\langle X \rangle\rangle \mid \forall p \in \mathcal{A}, \ s \triangleright p = 0\}$$

**Proposition 5.4.1** *$\mathcal{V}(\mathcal{A})$ est une algèbre de mélange, stable par résiduel à gauche et fermée pour la topologie usuelle sur les séries formelles i.e*

$$s, t \in \mathcal{V}(\mathcal{A}) \implies s \sqcup t \in \mathcal{V}(\mathcal{A})$$

$$s \in \mathcal{V}(\mathcal{A}) \implies p \triangleleft s \in \mathcal{V}(\mathcal{A}), \quad \forall p \in K\langle X\rangle$$
$$(\forall n \in \mathbb{N},\ s_n \in \mathcal{V}(\mathcal{A})),\ s_n convergente \implies \lim_{n\longrightarrow\infty} s_n \in \mathcal{V}(\mathcal{A})$$

**Preuve:** résulte directement de la définition de $\mathcal{V}(\mathcal{A})$. □

### 5.4.1 Bases finies d'une algèbre de mélange $\mathcal{V}(\mathcal{A})$

Dans toute cette section, on supposera que l'algèbre de Lie $\mathcal{A}$ est un sous–espace vectoriel de $\mathcal{L}ie\langle X\rangle$ de codimension finie $d$. On considèrera des polynômes de Lie $\{P_1, P_2, \cdots, P_d\}$ dont les images par le morphisme canonique forment une base de $\mathcal{L}ie\langle X\rangle/\mathcal{A}$.

**Definition 5.4.1** *Nous dirons qu'un ensemble $\{Z_1, Z_2 \ldots, Z_d\}$ forme une base de $\mathcal{V}(\mathcal{A})$* **ssi** *tout élément de $\mathcal{V}(\mathcal{A})$ se décompose de façon unique comme une série entière sur les $Z_i$ i.e.*

$$s = \sum_{\alpha} s_\alpha Z^{\sqcup\alpha} \quad avec\ s_\alpha \in K$$

*où $\alpha$ est un multi-indice.*

Autrement dit, on a: $\mathcal{V}(\mathcal{A}) = K[\![Z_1, Z_2, \cdots, Z_d]\!]$.

**Théorème 5.4.1** *Si $\mathcal{A}$ est de codimension finie $d$, alors $\mathcal{V}(\mathcal{A})$ admet une base de dimension $d$.*

**Preuve:** Voir [30]
On considère la base $\mathcal{B}$ de l'algèbre $\mathcal{L}ie\langle X\rangle$ obtenue en complétant l'ensemble $\{P_1, P_2, \cdots, P_d\}$ par une base $\{P_{d+1} \cdots\}$ de $\mathcal{A}(g)$. Soit $PBW.\mathcal{B}$ la base de Poincaré–Birkoff–Witt associée à la base $\mathcal{B}$. Pour un multi-indice quelconque $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d)$ on pose

$$\begin{aligned} Z^{\sqcup\alpha} &= Z_d^{\sqcup\alpha_d} \sqcup \cdots \sqcup Z_1^{\sqcup\alpha_1} \\ P^\alpha &= P_d^{\alpha_d} \ldots P_2^{\alpha_2} P_1^{\alpha_1} \\ \alpha! &= \alpha_1!\alpha_2! \cdots \alpha_d! \end{aligned}$$

Les séries $\{Z_1, Z_2, \cdots, Z_d\}$ appartenant à la base duale de la base $PBW.\mathcal{B}$ sont définies comme suit:

$$\begin{aligned} Z_i \in \mathcal{V}(\mathcal{A}) & \qquad \text{pour } i = 1..d \\ < Z_i \mid \varepsilon > &= 0 \qquad \text{pour } i = 1..d \\ < Z_i \mid P_j > &= \delta_i^j \qquad \text{pour } i, j = 1..d \\ Z_i \mid P^\alpha &= 0 \qquad \text{pour } i = 1..d,\ |\alpha| > 1 \end{aligned}$$

Le théorème de Poincaré–Birkoff–Witt permet de prouver que

$$\forall s \in \mathcal{V}(\mathcal{A}), \quad s = \sum_{\alpha} \frac{< s \mid P^\alpha >}{\alpha!} Z^{\sqcup\alpha}$$

Les séries $Z_i$ forment donc une base de $\mathcal{V}(\mathcal{A})$. □

Nous allons donner une condition suffisante pour qu'un ensemble $\{Z_1, Z_2 \ldots Z_d\}$ forme une base de $\mathcal{V}(\mathcal{A})$.

**Proposition 5.4.2** *Soient $Z_1, Z_2 \dots Z_d$ des séries propres de $\mathcal{V}(\mathcal{A})$ vérifiant la condition d'orthogonalité:*

$$< Z_i | P_j >= \delta_i^j \quad \textit{pour } 1 \leq i, j \leq d$$

*Alors* $\mathcal{V}(\mathcal{A}) = K[[Z_1, Z_2, \cdots, Z_d]]$

**Preuve:** Voir [30]
On cherche des coefficients $s_\alpha \in K$ tels que

$$s = \sum_\alpha s_\alpha Z^{\sqcup\alpha}$$

Cette dernière condition est équivalente à

$$< s \mid P^\beta > = \sum_\alpha s_\alpha < Z^{\sqcup\alpha} \mid P^\beta > \tag{5.1}$$

pour un multi-indice $\beta$ quelconque.

Les propositions 5.2.1 et 5.2.2 impliquent que

$$< Z^{\sqcup\alpha} \mid P^\beta > = 0 \text{ si } |\alpha| > |\beta|$$
$$< Z^{\sqcup\alpha} \mid P^\beta > = 0 \text{ si } |\alpha| = |\beta| \text{ et } \alpha \neq \beta$$
$$< Z^{\sqcup\alpha} \mid P^\alpha > = \alpha!$$

Il s'ensuit que le système (5.1) est triangulaire et donc permet de calculer les coefficients $s_\alpha$.

Le système comportant un nombre infini d'équations, la décomposition d'un polynôme de $\mathcal{V}(\mathcal{A})$ dans la base $\{Z_1, Z_2 \dots Z_d\}$ n'est pas forcément polynomiale. □

# 6

# La réalisation minimale locale

## 6.1 Introduction

Nous donnons un algorithme efficace pour calculer une représentation d'état minimale d'un système dynamique analytique ($\Sigma$) donné par sa série génératrice $g$ lorsque celle-ci est polynomiale.

Ceci constitue par la–même une preuve constructive de la conjecture de G.Jacob restée indémontrée:

*Une série génératrice polynomiale admet une réalisation minimale polynomiale.*

On sait d'après M.Fliess [9] que la réalisation minimale analytique est unique à un changement de coordonnées près (difféomorphisme analytique) dans l'espace d'état.

La réalisation que nous calculons est canonique au sens où elle ne dépend que de l'ordre choisi pour comparer les lettres de l'alphabet de codage. Ce résultat est acquis en utilisant les techniques de "base–standard" appliquées aux espaces vectoriels de polynômes.

De nombreux chercheurs ont travaillé sur la réalisation minimale analytique et les problèmes qui lui sont liés comme l'observabilité et la contrôlabilité en automatique non linéaire. Citons entre autres Hermann et Krenner(1977) [11], J.H.Sussman(1977) [32], Jakubczyk(1980) [16] et M.Fliess(1983) [9].

C. Reutenauer(1986) dans [30] a donné une version très "syntaxique" de l'existence de la réalisation minimale en effectuant son calcul dans une certaine algèbre de mélange. En partant d'une série polynomiale, la réalisation qu'il obtient est analytique (mais pas forcément polynomiale) et la traduction informatique de sa preuve ne donne pas un algorithme qui finit en un nombre fini de pas.

Reprenant les travaux de C.Reutenauer, N.E. Oussous dans [24] a implanté, en utilisant le système de calcul formel Macsyma, un algorithme qui, par une méthode d'exploration systématique de toutes les solutions possibles[1], calcule une réalisation minimale polynomiale mais sans toutefois démontrer que celle–ci existe dans tous les cas.

L'algorithme que nous proposons, outre qu'il est prouvé, est beaucoup plus rapide. Ainsi, une série génératrice de 3 variables en degré 7 est réalisée en moins d'une minute sur un

[1] La méthode relève de l'algèbre linéaire.

PC-RT [2] avec le système de calcul formel SCRATCHPAD (voir la démonstration page 143).

## 6.2 Notations et définitions de base

### 6.2.1 Définition d'un système dynamique

Un système dynamique analytique $(\Sigma)$ est défini par la donnée [9]

1. d'un espace d'état $E = \mathbb{R}^n$,
2. d'un ensemble $\{Y_0, Y_1, \ldots, Y_{m-1}\}$ de $m$ champs de vecteurs analytiques dans $E$,
3. d'une fonction d'observation $h : E \longrightarrow \mathbb{R}$,
4. d'un état initial $q(0) \in E$.

Le système $(\Sigma)$ est commandé par $m$ fonctions temporelles $\{u_0(t), u_1(t), \ldots, u_{m-1}(t)\}$ continues par morceaux.

L'évolution de l'état $q = (q_0, q_1, \ldots, q_{n-1})$ du système est gouvernée par l'équation:

$$\dot{q}(t) = \sum_{i=0}^{m-1} u_i(t) Y_i(q)$$

La fonction temporelle $s : t \in \mathbb{R} \longrightarrow h(q(t)) \in \mathbb{R}$ désigne la sortie du système.

### 6.2.2 Comportement Entrée/Sortie

Le comportement Entrée/Sortie est défini par la fonctionnelle

$$u \longrightarrow s$$

On considère l'alphabet de codage totalement ordonné $X = \{x_0, x_1, \ldots, x_{m-1}\}$ ; à chacune de ces $m$ lettres $x_i$ est associé un champ de vecteurs $Y_i$ et une entrée $u_i(t)$. On sait d'après M.Fliess [8] que la sortie est donnée par

$$s(t) = \sum_{w \in X^*} \mathcal{E}_u(w)_{|_t} \mathcal{Y}(w) \circ h_{|_{q(0)}}$$

où

- $\mathcal{Y}$ est un morphisme de monoïdes associant à tout mot $w \in X^*$ un opérateur différentiel $Y_w$ de l'algèbre de Weyl obtenu par composition des dérivées de Lie $Y_i$.

  $\mathcal{Y}$ est défini par :

$$\begin{aligned} \mathcal{Y}(\varepsilon) &= 1 \\ \forall x_i \in X, \quad \mathcal{Y}(x_i) &= Y_i \\ \forall u, v \in X^*, \quad \mathcal{Y}(uv) &= \mathcal{Y}(u) \circ \mathcal{Y}(v) \end{aligned}$$

  Par linéarité, le morphisme $\mathcal{Y}$ se prolonge en un morphisme d'algèbres défini sur $\mathbb{R}\langle X \rangle$.

[2] ©IBM

- $\mathcal{E}_u$ désigne un calcul d'intégrale itérée definie récursivement par

$$\begin{aligned} \mathcal{E}_u(\varepsilon) &= 1, \qquad \forall u \\ \mathcal{E}_u(wx_i)_{|_t} &= \int_0^t \mathcal{E}_u(w)_{|_\tau} u_i(\tau) d\tau, \qquad \forall w \in X^* \end{aligned}$$

La série génératrice de $(\Sigma)$ est une série en variables non commutatives définie par

$$g = \sum_{w \in X^*} \mathcal{Y}(w) \circ h_{|_{q(0)}} w$$

La sortie du système s'exprime sous la forme :

$$s(t) = \mathcal{E}_u(g)_{|_t}$$

### 6.2.3 Le problème posé

*Etant donnée une série formelle $g \in \mathbb{R}\langle\langle X \rangle\rangle$, calculer un système dynamique analytique $(\Sigma)$ dont la dimension de l'espace d'état est minimale et qui admet $g$ comme série génératrice.*

D'un point de vue effectif, ce problème est mal posé. Il convient d'abord de disposer d'une représentation finie de la série $g$, ce qui est possible pour certaines classes de séries comme les polynômes ou les séries rationnelles. D'autre part, il convient de se limiter à un ensemble de fonctions que l'on sait coder avec un nombre fini de symboles[3], ce qui n'est pas le cas des fonctions analytiques.

Ces considérations bassement matérielles expliquent peut-être pourquoi la théorie de la réalisation minimale analytique locale, bien que réglée "mathématiquement" est finalement peu utilisée par les praticiens de l'automatique non linéaire.

Dans cette section, nous supposerons que la série génératrice $g$ est un polynôme à coefficients dans un corps $K$ de caractéristique nulle. Pratiquement, $K$ sera souvent le corps $\mathbb{Q}$ car les ensembles $\mathbb{R}$ ou $\mathbb{C}$ ne sont pas effectifs.

## 6.3 Etude théorique

Pour simplifier, on supposera que le système dynamique que l'on désire réaliser a comme état initial $q(0) = 0$.

### 6.3.1 Série produite différentiellement

Une série $g \in K\langle\langle X \rangle\rangle$ est dite produite différentiellement [9] **ssi** il existe $m$ champs de vecteurs formels et une série formelle $h$ en $n$ variables commutatives qui donnent g:

$$g = \sum_{w \in X^*} \mathcal{Y}(w) \circ h_{|_0} w \tag{6.1}$$

[3] Un tel ensemble est nécessairement dénombrable.

Une série $g \in K\langle\langle X\rangle\rangle$ est dite convergente **ssi**

$$\exists C, M \in \mathbb{R},\ \forall w \in X^*, \qquad |< g|w >| \ \leq\ C|w|!\, M^{|w|}$$

Dans notre étude, nous en resterons au cas formel car, pour une série $g$ polynomiale, les problèmes de convergence ne se posent pas.

#### 6.3.1.1 Rang de Lie d'une série formelle

Par définition, le rang de Lie [9] d'une série quelconque $g$ est la dimension de l'espace vectoriel $g \triangleright \mathcal{L}ie\langle X\rangle$ (voir section 4.9.5). Il est élémentaire que le rang de Lie d'un polynôme ou d'une série rationnelle est fini.

L'algèbre de commande est par définition l'algèbre de Lie engendrée par l'ensemble des champs de vecteurs $\{Y_i\}_{0\leq i\leq m-1}$. On la notera $\mathcal{Y}(\mathcal{L}ie\langle X\rangle)$.

**Definition 6.3.1** *Le rang $r_q$ de l'algèbre de commande, défini en un point $q$ de l'espace d'état, est calculé après évaluation en $q$ des composantes des champs de vecteurs :*

$$r_q \ = \ \mathrm{rang}\{V_{|_q} \mid V \in \mathcal{Y}(\mathcal{L}ie\langle X\rangle)\}$$

Ce rang est donc majoré par la dimension $n$ de l'espace d'état.

**Proposition 6.3.1**
*Toute série produite différentiellement a un rang de Lie fini qui est majoré par le rang (en $q = 0$) de l'algèbre de Lie de commande.*

**Preuve:** Soit $p$ un polynôme de Lie quelconque ; on sait que l'opérateur différentiel $Y_p = \mathcal{Y}(p)$ est une dérivée de Lie. Il s'exprime donc comme une combinaison linéaire des $n$ dérivations partielles $\frac{\partial}{\partial q_i}$:

$$Y_p \ = \ \sum_{i=0}^{n-1} \Theta_i(q)\frac{\partial}{\partial q_i} \qquad \text{avec } \Theta_i \in K[\![q_0, q_1, \ldots, q_{n-1}]\!]$$

Le calcul du résiduel à droite de $g$ par $p$ donne :

$$\begin{aligned} g \triangleright p \ &= \ \sum_{w\in X^*} < g|pw > w \\ &= \ \sum_{w\in X^*} Y_p \circ Y_w \circ h_{|_0} w \\ &= \ \sum_{w\in X^*} Y_{p|_0} \circ Y_w \circ h_{|_0} w \end{aligned}$$

Soit $\{A_0, A_1, \ldots, A_{r-1}\}$ une base (au point $q = 0$) de l'algèbre de commande ; on a :

$$\begin{aligned} Y_{p|_0} \ &= \ \sum_{i=0}^{r-1} \alpha_i A_i \qquad \text{avec } \alpha_i \in K, i = 0..r-1 \\ g \triangleright p \ &= \ \sum_{w\in X^*} \left(\sum_{i=0}^{r-1} \alpha_i A_i\right) \circ Y_w \circ h_{|_0} w \\ &= \ \sum_{i=0}^{r-1} \alpha_i \left(\sum_{w\in X^*} A_i \circ Y_w \circ h_{|_0} w\right) \end{aligned}$$

$$= \sum_{i=0}^{r-1} \alpha_i T_i$$

En posant pour $i = 0..r-1$ :

$$T_i = \sum_{w \in X^*} A_i \circ Y_w \circ h_{|_0} w$$

On constate donc que $g \triangleright \mathcal{L}ie\langle X \rangle$ est un $K$–espace vectoriel engendré par les séries $T_i$. □

### 6.3.2 Le théorème de M.Fliess

Soit $(\Sigma)$ une réalisation de $g$, $d$ son rang de Lie, $n$ la dimension de l'espace d'état et $r$ le rang de l'algèbre de commande en $q = 0$. D'après la proposition 6.3.1, on a forcément:

$$d \leq r \leq n$$

Lorsqu'une réalisation est minimale, ces trois rangs sont égaux.

**Théorème 6.3.1 (Fliess [9])** - *Une série convergente est réalisable* **ssi** *son rang de Lie est fini. Le rang de Lie d correspond à la dimension minimale de toutes ses représentations différentielles.*

*La réalisation minimale est unique à un difféomorphisme analytique près sur les coordonnées locales* $(q_0, \ldots, q_{d-1})$.

**Remarque:** Il est clair que l'on peut toujours, en partant d'un système $(\Sigma)$, effectuer un changement de coordonnées locales ; on obtient un "nouveau" système dynamique $(\bar{\Sigma})$ ayant le même comportement Entrée/Sortie et la même série génératrice que $(\Sigma)$ en posant pour un difféomorphisme quelconque $\varphi : E \longrightarrow E$ :

$$\begin{aligned} \bar{q}(0) &= \varphi(q(0)) \\ \bar{h} &= h \circ \varphi^{-1} \\ \forall i (i = 0..m-1), \quad \bar{Y}_i &= (d\varphi) \circ Y_i \end{aligned}$$

Du point de vue de l'informaticien, il lui faut faire le choix d'une réalisation parmi une infinité de réalisations possibles, ce qui n'est pas si facile. Il serait intéressant de disposer d'une réalisation qui soit "canonique".

### 6.3.3 Interprétation dans l'algèbre de mélange des séries formelles

Soit $(\Sigma)$ un système dynamique de dimension $n$. On considère l'ensemble des séries entières $\mathcal{H} = K[\![q_0, \ldots, q_{n-1}]\!]$. On construit une interprétation $\sigma : \mathcal{H} \longrightarrow K\langle\langle X \rangle\rangle$ en posant [8]:

$$\forall f \in \mathcal{H}, \qquad \sigma(f) = \sum_{w \in X^*} \mathcal{Y}(w) \circ f_{|_0} w$$

Il est clair que l'interprétation $\sigma$ est une application linéaire.

On a les propriétés suivantes [8]:

$\forall f, f_1, f_2 \in \mathcal{H}$,

$$\begin{cases} \sigma(f_1 \cdot f_2) &= \sigma(f_1) \sqcup\!\sqcup \sigma(f_2) \\ \sigma(Y_i \circ f) &= x_i \triangleleft \sigma(f) \quad , i = 0..m-1 \\ \sigma(f_{|0}) &= < \sigma(f) | \varepsilon > \end{cases}$$

On constate donc que les dérivées de Lie $Y_i$ opérant sur $\mathcal{H}$ s'interprètent comme des calculs de résiduels à gauche par les lettres $x_i \in X$ sur les séries de $K\langle\langle X \rangle\rangle$.

#### 6.3.3.1 Interprétation des composantes des champs de vecteurs

Les composantes $\Theta_i^j$ des champs de vecteurs sont définies par

$$\forall i = 0..m-1, \qquad Y_i = \sum_{j=0}^{n-1} \Theta_i^j \frac{\partial}{\partial q_j}$$

On a alors:

$$\Theta_i^j = Y_i \circ q_j$$

qui s'interprète en

$$\sigma(\Theta_i^j) = x_i \triangleleft Z_j \tag{6.2}$$

en posant $Z_j = \sigma(q_j)$ pour $j = 0..n-1$.

### 6.3.4 Algorithme de C. Reutenauer

On considère l'algèbre de Lie

$$\mathcal{A}(g) = \{p \in \mathcal{L}ie\langle X \rangle \mid g \triangleright p = 0\}$$

et l'algèbre de mélange qui s'annule sur $\mathcal{A}(g)$

$$\mathcal{V}(\mathcal{A}(g)) = \{s \in K\langle\langle X \rangle\rangle \mid s \triangleright \mathcal{A}(g) = 0\}$$

On a: $g \triangleright \mathcal{L}ie\langle X \rangle \simeq \mathcal{L}ie\langle X \rangle / \mathcal{A}(g)$ donc $\mathcal{A}(g)$ est de codimension finie, égale au rang de Lie de $g$.

La proposition 5.4.1 s'applique donc: $\mathcal{V}(\mathcal{A}(g))$ est une algèbre de mélange, stable par résiduel à gauche et fermée pour la topologie usuelle sur les séries formelles. De plus, il est clair que $g \in \mathcal{V}(\mathcal{A}(g))$.

Le théorème 5.4.1 affirme que $\mathcal{V}(\mathcal{A}(g))$ admet une base finie. La difficulté vient du fait que rien ne prouve que cette base (construite par dualité) soit toujours formée de polynômes.

#### 6.3.4.1 Calcul d'une base polynomiale de $\mathcal{V}(\mathcal{A}(g))$

Soient $\{P_1, P_2, \cdots, P_d\}$ des polynômes de Lie tels que $\{g \triangleright P_1, g \triangleright P_2, \cdots, g \triangleright P_d\}$ forment une base de $g \triangleright \mathcal{L}ie\langle X \rangle$.

D'après le th. 5.4.2, il suffit de calculer des polynômes propres $\{Z_i \in \mathcal{V}(\mathcal{A}(g)) \mid i = 1..d\}$ vérifiant la condition d'orthogonalité:

$$< Z_i \mid P_j > = \delta_i^j \qquad \text{pour } i, j = 1..d \tag{6.3}$$

Les polynômes $Z_i$ s'obtiennent facilement par la formule:

$$Z_i = Q_i \triangleleft g - < Q_i | g > \qquad \text{pour } i = 1..d$$

où $Q_1, Q_2, \ldots, Q_d$ désignent des polynômes de $K\langle X \rangle$ définis par la relation d'orthogonalité:

$$< Q_j \mid g \triangleright P_i > \; = \; \delta_i^j \qquad \text{pour } i,j = 1..d$$

l'indépendance des polynômes $g \triangleright P_i$ assurant l'existence des polynômes $Q_j$.

### 6.3.4.2 Principe de l'algorithme

L'idée fondamentale de C.Reutenauer est de construire la réalisation minimale $(\Sigma)$ en prenant pour espace d'état $E = K^d$ et en interprétant les séries de $\mathcal{V}(\mathcal{A}(g))$ comme des séries entières définies sur $E$.

L'algorithme de principe qui en résulte est le suivant:

1. Calculer une base de $\mathcal{V}(\mathcal{A}(g))$
2. Décomposer $g$ dans la base $Z_i$ ; on obtient la fonction d'observation par simple traduction:

$$g = \sum_\alpha g_\alpha Z^{\sqcup\alpha} \implies h = \sum_\alpha g_\alpha q^\alpha$$

où $\alpha$ est un multi-indice.

3. Décomposer $x_i \triangleleft Z_j$ dans la base $Z_i$ ; on obtient d'après (6.2) les composantes des champs de vecteurs $\Theta_i^j$ par simple traduction:

$$\forall i,j \in 1..d, \qquad x_i \triangleleft Z_j = \sum_\alpha c_{i,\alpha}^j Z^{\sqcup\alpha} \implies \Theta_i^j = \sum_\alpha c_{i,\alpha}^j q^\alpha$$

où $\alpha$ est un multi-indice.

La difficulté vient du fait que l'algorithme de décomposition des éléments de $\mathcal{V}(\mathcal{A}(g))$ dans la base $Z_i$ n'est pas effectif ; il nécessite l'inversion d'un système linéaire triangulaire, potentiellement infini (voir preuve du th.5.4.2).

## 6.4 Calcul effectif

### 6.4.1 Condition triangulaire

Soit $g$ le polynôme à réaliser. Notre but est de construire une réalisation $(\Sigma)$ où la fonction d'observation et les composantes des champs de vecteurs sont des polynômes.

A cette fin, nous allons imposer sur les "coordonnées locales" $Z_i$ une condition plus forte que la simple relation d'orthogonalité définie en (6.3).

Nous dirons que le système $\{Z_i, P_j \mid i,j = 1..d\}$ où les polynômes $Z_i \in \mathcal{V}(\mathcal{A}(g))$ sont sans terme constant, vérifie la condition triangulaire **ssi** la matrice $Z_i \triangleright P_j$ est triangulaire ie.

$$\begin{cases} Z_i \triangleright P_j \; = \; 0 & i > j, \; i,j = 1..d \\ Z_i \triangleright P_i \; = \; 1 & i = 1..d \end{cases}$$

### 6.4.2 Calcul d'un système triangulaire

On considère des polynômes de Lie $\{P_1, P_2, \ldots, P_d\}$ tels que $\{g \triangleright P_1, g \triangleright P_2, \ldots, g \triangleright P_d\}$ constituent la base standard $G$ complètement réduite de l'espace vectoriel $g \triangleright \mathcal{L}ie\langle X\rangle$. Cette base-standard $G$ est unique une fois fixé l'ordre sur les mots de $X^*$, ici l'ordre lexicographique par longueur.

Soient $(w_1, w_2, \ldots, w_d)$ la liste croissante des mots de tête des polynômes de $G$. On sait d'après 4.7.2.1 que

$$\mathrm{lt}(G) \;=\; \mathrm{lt}\,(g \triangleright \mathcal{L}ie\langle X\rangle)$$

Ces mots que nous appelerons *mots caractéristiques* ne dépendent donc que de l'ordre choisi sur l'alphabet de codage $X$. On a par construction:

$$w_j \;=\; \mathrm{lt}(g \triangleright P_j) \qquad \text{pour } j = 1..d$$
$$w_i \;>\; w_j \qquad \text{pour } i > j,\ i,j = 1..d$$

La construction de la base-standard revient à triangulariser la matrice de Lie-Hankel (voir 6.5.2.1).

On construit les polynômes $Z_i$ par la formule:

$$Z_i \;=\; w_i \triangleleft g \;-\; < w_i | g > \qquad \text{pour } i = 1..d$$

Par construction, les polynômes $Z_i$ sont propres (sans terme constant).

Montrons que la condition triangulaire est satisfaite. On a:

$$\begin{aligned} Z_i \triangleright P_j \;&=\; (w_i \triangleleft g) \triangleright P_j \\ &=\; w_i \triangleleft (g \triangleright P_j) \end{aligned}$$

Par construction, si $i > j$, le mot $w_i$ est strictement plus grand (ordre lexicographique par longueur) que tous les mots figurant dans le polynôme $g \triangleright P_j$ donc $w_i \triangleleft (g \triangleright P_j) = 0$ et par suite $Z_i \triangleright P_j = 0$. □

### 6.4.3 Décomposition d'un polynôme

#### 6.4.3.1 Idée intuitive de l'algorithme

Soit $\{Z_i, P_j\}$ un système triangulaire et $s \in \mathcal{V}(\mathcal{A}(g))$ un polynôme que l'on cherche à décomposer dans la base $Z_i$; supposons que l'on ait:

$$s \;=\; A \sqcup Z_1^{\sqcup 2} + B \sqcup Z_1 + C$$

où $A, B, C \in \mathcal{V}(\mathcal{A}(g))$ et s'expriment uniquement en fonction de $Z_2..Z_d$.

On peut calculer $A, B, C$ en calculant les résiduels de $s$ par $P_1$. Les calculs donnent compte-tenu de la condition triangulaire:

$$\begin{cases} s \triangleright P_1 \;=\; 2A \sqcup Z_1 + B \\ s \triangleright P_1^2 \;=\; 2A \\ s \triangleright P_1^3 \;=\; 0 \end{cases}$$

On calcule d'abord $A$, puis par différence $B$, puis ensuite $C$. Il reste à décomposer $A, B, C$ dans la base $Z_2 \cdots Z_d$ en calculant les résiduels par $P_2$ etc... Cet algorithme est donc naturellement récursif.

**Remarque:** l'exemple est trompeur car il s'agit précisément de démontrer que le polynôme $s$ se décompose de façon polynomiale sur la base $\{Z_i\}$.

### 6.4.3.2 Algorithme de base

Cet algorithme **xtaylor** [4] décompose un polynôme quelconque $s$ de $\mathcal{V}(\mathcal{A}(g))$ dans la base $\{Z_i, i = 1..d\}$. Le résultat est un polynôme en variables commutatives de $K[q_1, q_2, \ldots, q_d]$ où $q_1, ..., q_d$ sont des lettres codant les polynômes $Z_1, \ldots, Z_d$.

Initialement, la fonction **xtaylor**$(s, lp, lz, lq)$ est appelée avec $lp = (P_j)_{j=1..d}$ et $lz = (Z_i)_{i=1..d}$, le système $\{Z_i, P_j\}$ étant supposé triangulaire ; $lq$ désigne la liste des lettres $(q_1, \ldots, q_d)$.

```
xtaylor (s, lp, lz, lq): PolynomeCommutatif ==
  si s est constant alors
     retourner(s)
  fsi
  si lp = ∅ alors
     erreur: cela ne peut pas se produire !!!
  fsi
  Initialisation:
     i := 0
     p: PolynomeDeLie := lp.first
     z: PolynomeNonCommutatif := lz.first
     q: PolynomeCommutatif := lq.first
     r: PolynomeNonCommutatif := s
     nr: PolynomeNonCommutatif := r ▷ p
  tq nr ≠ 0 faire
     r := nr ; nr := r ▷ p
     i := i + 1
  ftq
  resultat: PolynomeCommutatif :=
  (1/i!) q^i xtaylor(r, lp.rest, lz.rest, lq.rest) + xtaylor(s − (1/i!) z^⧢i ⧢ r, lp, lz, lq)
  retourner(resultat)
```

**Notations:** $l$.first et $l$.rest correspondent aux primitives (car $l$) et (cdr $l$) du langage LISP.

### Preuve de correction

L'invariant du "tant que" est $r = s \triangleright p^i$. En sortie du "tant que", on a $r \triangleright p = 0$ .

On vérifie facilement qu'au cours des appels récursifs:
- le premier polynôme de Lie de la liste $lp$ est $P_k$
- le premier polynôme non commutatif de $lz$ est $Z_k$
- le premier polynôme commutatif de $lq$ est $q_k$

pour $k = 1..d$, le cas où les trois listes sont vides faisant exception.

Démontrons que la fonction **xtaylor**$(s, lp, lz, lq)$ est correcte si le polynôme $s \in \mathcal{V}(\mathcal{A}(g))$ vérifie

[4] Je l'ai appelé ainsi parce qu'il reconstitue un polynôme à partir de ses dérivées (de Lie).

l'assertion d'entrée:

$$\forall i,\ 1 \leq i < k, \quad s \triangleright P_i \ = \ 0$$

Il est aisé de vérifier que cette assertion est encore vraie lors des deux appels récursifs de la fonction **xtaylor**.

Lorsque $k = d + 1$, les trois listes $lp, lz, lq$ sont vides. Le polynôme $s \in \mathcal{V}(\mathcal{A}(g))$ vérifie donc l'assertion:

$$\forall i,\ 1 \leq i \leq d, \quad s \triangleright P_i \ = \ 0 \tag{6.4}$$

La preuve de correction est terminée si on démontre que $s$ est constant car il est trivial de vérifier que dans ce cas, la fonction **xtaylor**$(s, lp, lz, lq)$ est correcte.

On peut construire une base $\mathcal{B}$ de $\mathcal{L}ie\langle X\rangle$ en complétant la base $\{P_1, P_2, \cdots, P_d\}$ par une base de $\mathcal{A}(g)$. Puisque $s \in \mathcal{V}(\mathcal{A}(g))$, on a $s \triangleright \mathcal{A}(g) = 0$. On en déduit d'après 6.4 que

$$s \triangleright \mathcal{L}ie\langle X\rangle = 0$$

Ceci démontre que $s$ est constant. □

**Preuve de terminaison**

Ne présente pas de difficulté. □

L'algorithme présenté est volontairement simple ; il peut être facilement optimisé en mémorisant les résiduels $s \triangleright p^i$ calculés dans la boucle **tq** $nr \neq 0$ ... **ftq**. L'appel récursif double est alors évité.

**Théorème 6.4.1 (Jacob,Oussous,Petitot (1991))** - *Une série génératrice polynomiale admet une réalisation minimale analytique où la fonction d'observation et les composantes des champs de vecteurs sont des polynômes.*

**Preuve:** l'algorithme développé dans cette section en est une preuve constructive. □

### 6.4.3.3 Un mini-exemple pour comprendre

Soit la série $g$ suivante :

$$g \ = \ x_0x_1x_0x_1 + x_1x_0x_1$$

On construit la base de Lyndon jusqu'au degré 4 (degré de $g$). On obtient la liste :

$$\{x_0,\ \ x_1,\ \ [x_0,x_1],\ \ [x_0,[x_0,x_1]],\ \ [[x_0,x_1],x_1],$$
$$[x_0,[x_0,[x_0,x_1]]],\ \ [x_0,[[x_0,x_1],x_1]],$$
$$[[[x_0,x_1],x_1],x_1]\}$$

En développant les crochets de Lie, cette même base devient :

$$\Big\{x_0,\ x_1,\ x_0x_1 - x_1x_0,\ x_0^2x_1 - 2x_0x_1x_0 + x_1x_0^2,$$
$$x_0x_1^2 - 2x_1x_0x_1 + x_1^2x_0,$$
$$x_0^3x_1 - 3x_0^2x_1x_0 + 3x_0x_1x_0^2 - x_1x_0^3,$$
$$x_0^2x_1^2 - 2x_0x_1x_0x_1 + 2x_1x_0x_1x_0 - x_1^2x_0^2,$$
$$x_0x_1^3 - 3x_1x_0x_1^2 + 3x_1^2x_0x_1 - x_1^3x_0\Big\}$$

On calcule un système générateur de $g \triangleright \mathcal{L}ie\langle X\rangle$ :

$$\{x_1x_0x_1, x_0x_1, -x_1 + x_0x_1, -2x_1, -2, 0, -2, 0\}$$

La base standard de $g \triangleright \mathcal{L}ie\langle X\rangle$ est formé des polynômes :

$$\{x_1x_0x_1,\ x_0x_1,\ x_1,\ 1\}$$

dont les mots de tête $(w_i)$ sont :

$$\{x_1x_0x_1,\ x_0x_1,\ x_1,\ 1\}$$

Les polynômes de Lie $(P_i)$ associés :

$$\left\{-\frac{1}{2}[[x_0, x_1]x_1],\ -\frac{1}{2}[x_0, [x_0, x_1]],\ [x_1],\ [x_0]\right\}$$

Les coordonnées locales $(Z_i = w_i \triangleleft g - \langle w_i|g\rangle)$ :

$$\{x_0x_1x_0x_1 + x_1x_0x_1, x_0x_1x_0 + x_1x_0, x_0x_1 + x_1, x_0\}$$

On vérifie bien que la matrice $Z_i \triangleright P_j$ est triangulaire :

$$\begin{bmatrix} 1 & x_1 & x_0x_1 & x_1\,x_0x_1 \\ 0 & 1 & x_0 & x_1x_0 \\ 0 & 0 & 1 & x_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Calcul de la fonction d'observation :**

On a:

$$g = Z_0 \quad \text{d'où} \quad h = q_0$$

**Calcul des champs de vecteurs :**

Aux polynômes $x_0 \triangleleft Z_i$ :

$$\{0, x_0x_1 + x_1, 0, 1\}$$

correspond le champ de vecteurs :

$$Y_0 = q_2\frac{\partial}{\partial q_1} + \frac{\partial}{\partial q_3}$$

Aux polynômes $x_1 \triangleleft Z_i$ :

$$\{x_0x_1x_0 + x_1x_0, 0, x_0 + 1, 0\}$$

correspond le champ de vecteurs :

$$Y_1 = q_1\frac{\partial}{\partial q_0} + (q_3 + 1)\frac{\partial}{\partial q_2}$$

## 6.5 Implantation en SCRATCHPAD

### 6.5.1 Le théorème de Radford et la condition triangulaire

L'algorithme **xtaylor**$(s, lp, lz, lq)$ basé sur la condition triangulaire est intéressant en soi. Il peut en effet servir à implanter la décomposition d'un polynôme $s \in K\langle X\rangle$ dans la base de Radford.

**Théorème 6.5.1 (Radford [28])** - *Les mots de Lyndon construits sur un alphabet $X$ forment une base de l'algèbre de mélange $K\langle X\rangle$.*

**Lemme 6.5.1** *Soit $\{l_i \mid i > 1\}$ l'ensemble des mots de Lyndon construits sur un alphabet $X$, ordonnés par ordre croissant (ordre lexicographique par longueur). La matrice $l_i \triangleright [l_j]$ est triangulaire ie.*

$$\begin{cases} l_i \triangleright [l_j] = 0 & si\ i < j \\ l_i \triangleright [l_j] = 1 & si\ i = j \end{cases}$$

**Preuve** Facile en utilisant le lemme 3.4.1. □

## 6.5.2 La taille des données

### 6.5.2.1 La matrice de Lie-Hankel

Rappelons que la matrice de Lie-Hankel [9] notée $H$ est définie par

$$H_p^w = < g \triangleright p \mid w >$$

où le polynôme $p$ décrit une base de $\mathcal{L}ie\langle X\rangle$ et où $w$ décrit l'ensemble $X^*$.

Lorsque $g$ est un polynôme, on pourra se restreindre à la tranche $|w| \leq$ degré$(g)$ et degré$(p) \leq$ degré$(g)$ car le reste est identiquement nul. Le nombre de colonnes correspondant à $|w| \leq$ degré$(g)$ croît de façon exponentielle par rapport au degré de $g$.

Une solution consiste à remarquer qu'en général, la matrice de Lie-Hankel est très creuse et qu'on peut, de ce fait supprimer un certain nombre de lignes ou de colonnes identiquement nulles mais celà prend un "certain temps". Une deuxième difficulté tient au fait que la matrice $H$ est naturellement indicée par des mots et non par des entiers comme il est d'usage (à tort) généralement pour les matrices. Or la bijection $X^* \longleftrightarrow \mathbb{N}$ est loin d'être négligeable en temps calcul.

Pour toutes ces raisons, je pense qu'il est préfèrable de ne pas construire la matrice de Lie-Hankel et d'utiliser en remplacement les algorithmes de bases-standard appliqués aux espaces vectoriels de polynômes.

### 6.5.2.2 La taille de la série génératrice

La démonstration E page 143 montre que le nombre de mots de la série génératrice croît très rapidement lorsque celle-ci est construite par des produits de mélange, ce qui est fort pénalisant pour le calcul de la réalisation minimale.

Cette difficulté est évitée en représentant $g$ dans une base de l'algèbre de mélange. Les deux bases classiques de cette algèbre sont la base de Radford (voir 6.5.1) et la base duale de la base $PBWL$ (voir 5.3). Le calcul du produit de mélange se ramène alors à un simple produit en variables commutatives et la taille des données augmente beaucoup moins vite.

Cet exemple illustre parfaitement le fait que la taille des données est assez éloignée de la "quantité d'information" qu'elles contiennent.

# 7

# Groupe de Lie d'une algèbre nilpotente

## 7.1 Motivation

Le problème du *motion-planning* consiste à calculer les entrées $u_i(t)$ qui font transiter un système dynamique $(\Sigma)$ d'un état initial $q_0$ à un état final donné $q$. Ce problème a été étudié par H.Sussmann et J.Lafférière [19] [33] pour un système sans dérive, lorsque l'algèbre des champs de vecteurs est nilpotente et en se limitant à des entrées constantes par morceaux.

Notre but est d'implanter un algorithme effectif de motion-planning dans le système de calcul formel SCRATCHPAD pour un système avec dérive et des entrées plus générales (polynomiales). G. Jacob a publié récemment deux articles dans ce sens [13].

L'algorithme, qui est en cours d'implantation, nécessite de nombreuses compétences en calcul formel et numérique (résolution d'équations algébriques, décompositions cylindriques, intégration formelle et algèbre non commutative).

On trouvera, dans ce chapitre, un exposé et une implantation effective des algorithmes de calcul dans le groupe de Lie d'une algèbre nilpotente ; celà nous permettra de calculer quelques formules utiles au motion-planning.

## 7.2 Décomposition en produit décroissant d'exponentielles

Soit $X$ un alphabet ordonné et $\mathcal{L}ie\langle X\rangle$ l'algèbre de Lie libre à coefficients dans un corps $K$.

On construit une algèbre nilpotente à l'ordre $n$ en considérant le quotient de $\mathcal{L}ie\langle X\rangle$ par l'idéal des polynômes de Lie engendré par les crochets de longueur strictement supérieure à $n$. Cette algèbre est isomorphe (en tant qu'espace vectoriel) à

$$\mathcal{L}ie_{\leq n}\langle X\rangle = \{p \in \mathcal{L}ie\langle X\rangle | degre(p) \leq n\}$$

On considère le groupe de Lie associé:

$$\mathcal{G} = \{e^p | p \in \mathcal{L}ie_{\leq n}\langle X\rangle\}$$

On choisit de représenter les polynômes de $\mathcal{L}ie_{\leq n}\langle X\rangle$ dans la base de Lyndon en se restreignant aux mots de Lyndon de longueur inférieure ou égale à $n$.

Soit $\{l_1, l_2 \ldots l_d\}$ une telle base ordonnée par ordre décroissant. Par définition, tout élément $g \in \mathcal{G}$ s'écrit de façon unique sous la forme:

$$g = e^{\alpha_1[l_1]+\alpha_2[l_2]+\cdots\alpha_d[l_d]}$$

**Théorème 7.2.1** *Tout élément du groupe de Lie se décompose de façon unique en produit décroissant des exponentielles de base i.e.*

$$g = e^{\xi_1[l_1]}e^{\xi_2[l_2]} \ldots e^{\xi_d[l_d]}$$

*avec* $l_1 > l_2 > \cdots > l_d$ *et* $\xi_1, \xi_2, \ldots \xi_d \in K$.

Le $d$-uplet $(\xi_1, \xi_2 \ldots \xi_d)$ sera appelé les *coordonnées de Lyndon* de $g$.

### 7.2.1 Représentation des éléments du groupe de Lie

Un élément $g$ du groupe de Lie $\mathcal{G}$ est une série en variables non commutatives:

$$\begin{aligned} g &= e^p \qquad \text{pour } p \in \mathcal{L}ie_{\leq n}\langle X\rangle \\ &= 1 + p + \frac{p^2}{2!} + \frac{p^3}{3!} + \cdots \end{aligned}$$

Il est licite de tronquer la série $g$ à l'ordre $n$ car les mots de longueur supérieure à $n$ n'interviennent pas dans le calcul de $p$ à partir de $g$

$$\begin{aligned} p &= \log(g) = \log(1 + g_1) \\ &= g_1 - \frac{g_1^2}{2} + \frac{g_1^3}{3} + \cdots + (-1)^{n+1}\frac{g_1^n}{n} \end{aligned}$$

avec $g_1 = g - 1$ qui est une série sans terme constant.

Nous pouvons donc représenter tout élément $g$ par un polynôme non commutatif dont le degré est inférieur ou égal à $n$. Le calcul du produit de deux éléments du groupe de Lie se ramène alors à un simple produit de deux polynômes de $K\langle X\rangle$.

La section suivante fera apparaître que la représentation dans la base $PBWL$ est avantageuse si l'on désire exprimer un élément $g \in \mathcal{G}$ en produit décroissant d'exponentielles.

### 7.2.2 Algorithme de décomposition en produit d'exponentielles

**Lemme 7.2.1** *Les coordonnées de Lyndon d'un élément $g \in \mathcal{G}$ sont les coefficients des polynômes de Lyndon figurant dans la décomposition de $g$ dans la base $PBWL$.*

**Preuve:** Pour simplifier l'exposé, supposons qu'il n'y ait que deux coordonnées $\xi_1$ et $\xi_2$ à calculer ; les calculs donnent:

$$\begin{aligned} g &= e^{\xi_1[l_1]}e^{\xi_2[l_2]} \quad \text{avec } l_1 > l_2 \\ &= (1 + \xi_1[l_1] + \frac{\xi_1^2[l_1]^2}{2!} + \cdots) \times (1 + \xi_2[l_2] + \frac{\xi_2^2[l_2]^2}{2!} + \cdots) \\ &= 1 + \xi_1[l_1] + \xi_2[l_2] + \underbrace{\xi_1\xi_2[l_1][l_2] + \cdots}_{(1)} \end{aligned}$$

La partie (1) qui contient uniquement des produits décroissants de polynômes de Lyndon, n'intervient pas dans l'identification des coordonnées de Lyndon de $g$.□

Les coordonnées de Lyndon de $g \in \mathcal{G}$ sont donc données par:

$$\xi_l = < g \mid S_l > \qquad \text{pour } l \in \mathcal{L}yndon\langle X\rangle, |l| \leq n$$

### 7.2.3 Algorithme de calcul de l'inverse d'un élément

Considérons le morphisme d'algèbre $\varrho : K\langle X\rangle \longrightarrow K\langle X\rangle$ défini sur les mots $w \in X^*$ par [14]

$$\varrho(w) = (-1)^{|w|}\tilde{w} \qquad \text{où } \tilde{w} \text{ désigne le miroir du mot } w.$$

**Lemme 7.2.2**

$$\forall g \in \mathcal{G}, \quad g^{-1} = \varrho(g)$$

**Preuve:** On vérifie facilement que $\varrho(e^p) = e^{\varrho(p)}$, or nous avons montré en 3.4.1 que

$$\forall p \in \mathcal{L}ie\langle X\rangle, \quad \varrho(p) = -p$$

donc $g^{-1} = e^{-p} = e^{\varrho(p)} = \varrho(g)$. □

## 7.3 Implantation en Scratchpad

Le constructeur de domaine `LieExponentials(vl,R,Order)` implante les calculs dans le groupe de Lie construit sur un alphabet `vl` et un anneau `R` passés en paramètres. Le degré de nilpotence correspond au paramètre `Order`.

Les éléments du groupe de Lie sont représentés par des polynômes du domaine `XPBWPolynomial` (voir 3.5.2). J'ai choisi d'afficher ces éléments sous forme de produits décroissants d'exponentielles car l'algorithme de calcul des coordonnées de Lyndon est très peu couteux.

On aurait pu adopter une autre représentation (par les polynômes de Lie) en utilisant la définition $g = e^p$ avec $p \in \mathcal{L}ie_{\leq n}\langle X\rangle$ ; dans ce cas, le produit de deux éléments du groupe de Lie (produit de Hausdorff) devient couteux.

```
)abbrev domain LEXP LieExponentials

LieExponentials(vl ,R, Order): public == private where
  EX     ==> Expression
  vl     : List EX
  VarSet ==> OrderedVarlist1(vl)
  PI     ==> PositiveInteger
  NNI    ==> NonNegativeInteger
  I      ==> Integer
  RN     ==> RationalNumber
  R      : Join(Ring, Module RN)
  Order  : PI
  LWORD  ==> LyndonWord(VarSet)
  LWORDS ==> List LWORD
  BASIS  ==> PoincareBirkoffWittLyndonBasis(VarSet)
  TERM   ==> Record(k:BASIS, c:R)
```

```
LTERMS ==> List(TERM)
LPOLY  ==> LiePolynomial(VarSet,R)
XDPOLY ==> XDistribuedPolynomial(VarSet,R)
PBWPOLY==> XPBWPolynomial(VarSet, R)
TERM1  ==> Record(k:LWORD, c:R)
EQ     ==> Equation(R)

public == Group with
  exp         : LPOLY -> $        -- exponentielle d'un polynome de Lie
  log         : $ -> LPOLY        -- logarithme
  ListOfTerms : $ -> LTERMS
  coerce      : $ -> XDPOLY
  coerce      : $ -> PBWPOLY
  mirror      : $ -> $            -- image miroir d'un element du groupe
  varList     : $ -> List VarSet  -- liste des variables
  LyndonBasis : () -> List LPOLY  -- liste des polynomes de base
  LyndonBasis : NNI -> LPOLY
  LyndonCoordinates: $ -> List TERM1
  identification: ($,$) -> List EQ

private == PBWPOLY add

  -- Representation
     Rep := PBWPOLY
  .....
```

(voir fichier de démonstration en annexe page 147)

## 7.4 Série de Chen et opérateur de transport

La série de Chen est une série en variables non commutatives calculable à chaque instant $t$ à partir de la donnée d'une entrée $u$ (voir 6.2.2). Elle est définie par par la formule:

$$S_u(t) = \sum_{w \in X^*} \mathcal{E}_u(w)_{|_t} w$$

H.J.Sussmann utilise souvent la définition équivalente suivante:

$$\begin{cases} \dot{S}_u(t) = S_u(t) \cdot \sum_{x \in X} u_x x \\ S_u(0) = 1 \end{cases}$$

Ree [29] a démontré que la série de Chen est l'exponentielle d'une série de Lie.

On sait d'après la formule fondamentale de M. Fliess (voir 6.2.2) que la sortie d'un système dynamique est donnée par la formule:

$$s(t) = \sum_{w \in X^*} \mathcal{E}_u(w)_{|_t} \mathcal{Y}(w) \circ h_{|_{q_0}}$$

de la forme $\mathcal{H}_u(t) \circ h_{|q_0}$ où $\mathcal{H}_u(t)$ est un opérateur différentiel appartenant au groupe de Lie et défini par [12]:

$$\mathcal{H}_u(t) = \sum_{w \in X^*} \mathcal{E}_u(w)_{|_t} \mathcal{Y}(w)$$

**Proposition 7.4.1 ([12])** *L'opérateur de transport de transport se factorise en un produit décroissant d'exponentielles:*

$$\mathcal{H}_u(t) = \prod_{l \in \mathcal{L}yndon\langle X \rangle} \exp\left(\mathcal{E}_u(S_l)_{|_t} \cdot \mathcal{Y}(Q_l)\right)$$

**Preuve rapide:** On peut considérer que $\mathcal{H}_u(t)$ est l'image de la série double $\sum_{w \in X^*} w \otimes w$ par le morphisme $\mathcal{E}_{u|_t} \otimes \mathcal{Y}$. La factorisation cherchée résulte de la factorisation de la série double vue à la section 5.3.2. □

Lorsque l'algébre de Lie des champs de vecteurs est nilpotente, ce produit est fini ; par définition, pour tous les mots de Lyndon $l$ de longueur supérieure à l'ordre de nilpotence $n$, on a: $\mathcal{Y}(Q_l) = 0$.

Dans la pratique, il est utile de mener les calculs de manière syntaxique, en ne supposant rien sur les champs de vecteurs (à part la nilpotence) et de considérer la série de Chen:

$$S_u(t) = \prod_{l \in \mathcal{L}yndon\langle X \rangle} \exp\left(\mathcal{E}_u(S_l)_{|_t} \cdot Q_l\right) \tag{7.1}$$

L'opérateur de transport $\mathcal{H}_u(t)$ se déduit alors de la série $S_u(t)$ par l'application du morphisme $w \longrightarrow \mathcal{Y}(w)$

## 7.4.1 Calcul effectif de la série de Chen

### 7.4.1.1 Entrée constante

On considère une entrée constante ( $u_0(t) = k_0, u_1(t) = k_1, \ldots$ ) ; la sortie du système dynamique est une intégrale du champ de vecteurs $k_0 Y_0 + k_1 Y_1 + \cdots$. Les opérateurs $\mathcal{H}_u(t)$ forment donc un groupe à un paramètre d'où la formule:

$$S_u(t) = e^{(k_0 x_0 + k_1 x_1 + \cdots)t}$$

### 7.4.1.2 Entrées concaténées

Soient $u$ et $v$ deux entrées définies respectivement sur les intervalles de temps $0..t_1$ et $0..t_2$. L'entrée $u\sharp v$ définie sur l'intervalle $0..t_1 + t_2$ désignera l'entrée obtenue par concaténation des entrées $u$ et $v$.

On dispose alors d'une formule classique [8]:

$$S_{u\sharp v}(t_1 + t_2) = S_u(t_1) \cdot S_v(t_2)$$

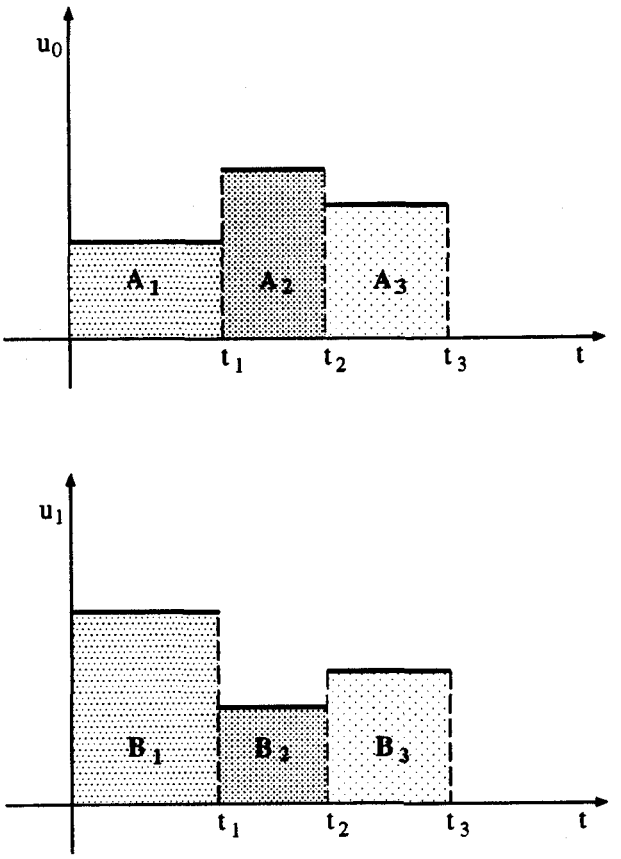



Figure 7.1 : Entrées constantes par morceaux

#### 7.4.1.3 Entrées quelconques

La formule (7.1) n'est effective que lorsque l'algèbre de commande est nilpotente. Dans ce cas, l'algorithme de construction des polynômes $S_l$ (voir 5.3.1) se traduit directement en algorithme de calcul de $\mathcal{E}_u(S_l)_{|t}$.

#### 7.4.1.4 Application: entrées constantes par morceaux

Soit $(u_0, u_1)$ une entrée comportant 3 paliers (voir figure 7.1).

On a alors:

$$S_u(t_3) = e^{A_1x_0+B_1x_1} \cdot e^{A_2x_0+B_2x_1} \cdot e^{A_3x_0+B_3x_1}$$

Les calculs en SCRATCHPAD, effectués pour un ordre de nilpotence $n = 3$, grâce à notre implantation des exponentielles de Lie donnent une série de la forme:

$$S_u(t_3) = e^{\xi_4[l_4]} \cdot e^{\xi_3[l_3]} \ldots e^{\xi_0[l_0]} \tag{7.2}$$

avec

$$\left\{\begin{array}{rcl@{\quad}rcl}
l_4 = x_1 & \longrightarrow & \xi_4 = & B_1+B_2+B_3 \\
l_3 = x_0x_1^2 & \longrightarrow & \xi_3 = & \frac{1}{6}A_1B_1^2+\frac{1}{2}A_1B_1B_2+\frac{1}{2}A_1B_1B_3+\frac{1}{2}A_1B_2^2+A_1B_2B_3 \\
 & & + & \frac{1}{2}A_1B_3^2+\frac{1}{6}A_2B_2^2+\frac{1}{2}A_2B_2B_3+\frac{1}{2}A_2B_3^2+\frac{1}{6}A_3B_3^2 \\
l_2 = x_0x_1 & \longrightarrow & \xi_2 = & \frac{1}{2}A_1B_1+A_1B_2+A_1B_3+\frac{1}{2}A_2B_2+A_2B_3+\frac{1}{2}A_3B_3 \\
l_1 = x_0^2x_1 & \longrightarrow & \xi_1 = & \frac{1}{6}A_1^2B_1+\frac{1}{2}A_1^2B_2+\frac{1}{2}A_1^2B_3+\frac{1}{2}A_1A_2B_2+A_1A_2B_3 \\
 & & + & \frac{1}{2}A_1A_3B_3+\frac{1}{6}A_2^2B_2+\frac{1}{2}A_2^2B_3+\frac{1}{2}A_2A_3B_3+\frac{1}{6}A_3^2B_3 \\
l_0 = x_0 & \longrightarrow & \xi_0 = & A_1+A_2+A_3
\end{array}\right.$$

#### 7.4.1.5 Application: entrées concaténées

Soient

$$S_u = e^{a_4[l_4]}e^{a_3[l_3]}e^{a_2[l_2]}e^{a_1[l_1]}e^{a_0[l_0]}$$
$$S_v = e^{b_4[l_4]}e^{b_3[l_3]}e^{b_2[l_2]}e^{b_1[l_1]}e^{b_0[l_0]}$$

Les formules suivantes permettent de multiplier deux séries de Chen lorsque l'algèbre de commande est nilpotente à l'ordre 3.
On pose

$$S_{u\sharp v} = e^{z_4[l_4]}e^{z_3[l_3]}e^{z_2[l_2]}e^{z_1[l_1]}e^{z_0[l_0]}$$

Les calculs en SCRATCHPAD donnent:

$$\left\{\begin{array}{rcl}
z_4 & = & b_4+a_4 \\
z_3 & = & \frac{1}{2}a_0b_4^2+a_2b_4+b_3+a_3 \\
z_2 & = & a_0b_4+b_2+a_2 \\
z_1 & = & \frac{1}{2}a_0^2b_4+a_0b_2+b_1+a_1 \\
z_0 & = & b_0+a_0
\end{array}\right.$$

On trouvera en annexe F page 151 les mêmes calculs effectués pour une algèbre de commande nilpotente à l'ordre 5.

## 7.5 Motion-planning: ce n'est qu'un début ...

On fait les hypothèses suivantes sur l'algèbre de commande:

1. Elle est nilpotente à l'ordre $n$.
2. Elle est de rang maximum en chaque point de l'espace d'état (le rang de l'algèbre de Lie de commande – voir 6.3.1.1 – est égal à la dimension de l'espace d'état).

Le problème est de calculer une entrée $u$ menant un système dynamique analytique ($\Sigma$) d'un état $q_0$ à un état $q$ donnés, en un temps quelconque $t$ non fixé.

### 7.5.1 Principe de l'algorithme

1. Calculer un élément $\mathcal{D}$ du groupe de Lie menant $(\Sigma)$ de $q_0$ à $q$.

2. Calculer une entrée $u$ définie sur l'intervalle de temps $[0..t]$ réalisant cet opérateur $\mathcal{D}$ ie. telle que :

$$\mathcal{D} = \mathcal{H}_u(t)$$

En pratique, les opérateurs $\mathcal{D}$ et $\mathcal{H}_u(t)$ sont définis par des séries de Chen exprimées en produits décroissants d'exponentielles ; il suffit donc d'identifier leurs coordonnées de Lyndon respectives.

### 7.5.2 Première partie de l'algorithme

On considère une trajectoire quelconque $(\gamma)$ allant de $q_0$ à $q$ en un temps quelconque $t$ (on peut supposer $t = 1$).

A tout instant $\tau$ pour $0 \leq \tau \leq t$, le vecteur vitesse $\dot{\gamma}(\tau)$ se décompose comme une combinaison linéaire de vecteurs $g_i$ de l'algèbre de Lie de commande ie.

$$\dot{\gamma}(\tau) = \sum_i \alpha_i(\tau) g_i \qquad \text{avec } \alpha_i(\tau) \in \mathbb{R} \tag{7.3}$$

Les composantes $\alpha_i(\tau)$ peuvent être considérées comme une commande $\alpha$ dans le système étendu aux champs $g_i$.

La formule (7.1) appliquée à cette entrée $\alpha$ permet de calculer une série de Chen $S_\alpha(t)$ définissant l'opérateur $\mathcal{D}$ ; une difficulté tient au fait qu'il est nécessaire d'utiliser des mots de Lyndon $L$ construits sur un alphabet dont les lettres $l$ sont elles-mêmes des mots de Lyndon ($l \in \mathcal{L}yndon\langle X \rangle$).

**Exemple:** système $(\Sigma)$ nilpotent à l'ordre 3 et $X = \{x_0, x_1\}$.

Il y a 5 champs de vecteurs non nuls $g_0 \ldots g_4$ dans le système étendu, codés respectivement par les mots de Lyndon:

$$l_0 = x_0 \qquad l_1 = x_0^2 x_1 \qquad l_2 = x_0 x_1 \qquad l_3 = x_0 x_1^2 \qquad l_4 = x_1$$

L'opérateur $\mathcal{D}$ calculé correspond à une série de Chen de la forme:

$$S_\alpha(t) = \prod_{i=9}^{0} e^{Z_i[L_i]} \qquad \text{avec } Z_i = \mathcal{E}_\alpha(S_{L_i})_{|_t}$$

On pourra remarquer que certains mots de Lyndon $L$ ne sont pas utiles car, compte–tenu de

la nilpotence, $\mathcal{Y}([L]) = 0$. Finalement il ne reste que 10 mots utiles à savoir:

$$\begin{cases} L_9 = l_4 \\ L_8 = l_3 \\ L_7 = l_2 l_4 \\ L_6 = l_2 \\ L_5 = l_1 \\ L_4 = l_0 l_4^2 \\ L_3 = l_0 l_4 \\ L_2 = l_0 l_2 \\ L_1 = l_0^2 l_4 \\ L_0 = l_0 \end{cases}$$

Il convient d'exprimer cette série comme un produit décroissant de 5 exponentielles telles que:

$$S_\alpha(t) = e^{z_4[l_4]} e^{z_3[l_3]} \dots e^{z_0[l_0]}$$

Les calculs effectués en SCRATCHPAD donnent:

$$\begin{cases} l_4 = x_1 & \longrightarrow & z_4 = Z_9 \\ l_3 = x_0 x_1^2 & \longrightarrow & z_3 = Z_8 + Z_7 + Z_4 \\ l_2 = x_0 x_1 & \longrightarrow & z_2 = Z_6 + Z_3 \\ l_1 = x_0^2 x_1 & \longrightarrow & z_1 = Z_5 + Z_2 + Z_1 \\ l_0 = x_0 & \longrightarrow & z_0 = Z_0 \end{cases}$$

On obtient des formules qui sont linéaires en $Z$ mais ceci est un heureux hasard qui dépend fortement de la façon de numéroter les mots de Lyndon $\{x_1, x_0 x_1^2, \dots, x_0\}$.

**Remarque:** La première partie de l'algorithme fournit donc un ensemble de 5 coordonnées de Lyndon $\{z_0, \dots, z_4\}$ qui ne sont pas uniques car la trajectoire $(\gamma)$ reliant les deux états donnés $q_0$ et $q$ est arbitraire.

### 7.5.3 Deuxième partie de l'algorithme

Cette partie repose sur une identification de deux séries de Chen données par leurs coordonnées de Lyndon. L'identification fournit en général un système d'équations algébriques (les inconnues sont réelles) qu'il convient de résoudre, soit symboliquement en construisant un système triangulaire (algorithme de Wu implanté par D. Lazard), soit numériquement lorsque la résolution symbolique s'avère trop coûteuse.

**Exemple:** On suppose le système $(\Sigma)$ nilpotent à l'ordre 3, les entrées cherchées étant constantes par morceaux. On obtient après identification, grâce aux formules (7.2), le système algèbrique suivant:

$$\begin{cases} A_1 + A_2 + A_3 = z_0 \\ \frac{1}{2} A_1 A_3 B_3 + \frac{1}{6} A_2^2 B_2 + \frac{1}{2} A_2^2 B_3 + \dots = z_1 \\ \dots = \dots \\ B_1 + B_2 + B_3 = z_4 \end{cases}$$

les coordonnées de Lyndon $\{z_0, \dots, z_4\}$ étant fournies par la première partie de l'algorithme.

**Remarque 1:** Il n'y a aucune raison pour que la trajectoire obtenue à partir de l'entrée constante par morceaux $u$ soit identique à la trajectoire d'essai $\gamma$ utilisée pour calculer l'opérateur $\mathcal{D}$. L'égalité $\mathcal{D} = \mathcal{H}_u(t)$ implique seulement que les trajectoires ont même point de départ et même point d'arrivée.

**Remarque 2:** Le lecteur vérifiera que la technique d'identification proposée peut s'étendre facilement à des entrées polynomiales.

#### 7.5.3.1 Cas avec dérive

Affirmer que le système $(\Sigma)$ comporte une dérive $Y_0$ revient à supposer que l'entrée correspondante $u_0$ est constante:

$$\forall \tau \in [0,t], \qquad u_0(\tau) \;=\; 1$$

Il convient donc de poser les conditions de signe $A_1, A_2, A_3 \geq 0$. La coordonnée de Lyndon $z_0 = A_1 + A_2 + A_3$ représente alors le *temps total* pour aller de l'état initial $q_0$ à l'état final $q$.

L'identification proposée conduit alors à résoudre un système d'équations algébriques où les inconnues sont réelles, certaines étant supposées positives. Il existe actuellement des algorithmes de résolution relevant, soit du calcul formel, soit du calcul numérique.

Par contre, l'existence et le calcul d'une *bonne* trajectoire d'essai $\gamma$ (aboutissant à un système d'équations ayant des solutions) est un problème auquel on ne peut apporter que des réponses partielles.

### 7.5.4 Conclusion

L'algorithme du Motion-planning est un exercice assez difficile de calcul formel ; il se peut que certains calculs ne puissent être effectués que numériquement, la complexité des algorithmes en calcul formel étant prohibitive. Cependant, la partie relevant de l'algèbre non commutative (manipulation des exponentielles de Lie) est terminée et les premières simulations effectuées par F.Boussemard [3] sont encouragentes.

# 8

# Conclusion

Le premier résultat pratique de ce travail est que le problème de la réalisation minimale d'une série génératrice polynomiale est résolu **effectivement** par un algorithme prouvé et ... calculant le résultat en un nombre fini d'étapes.

Des éléments (calcul du rang de Lie) sont apportés pour aborder la question de la réalisation minimale d'une série rationnelle mais des difficultés subsistent pour terminer complètement l'algorithme ; en gros, il nous manque un théorème précisant que la réalisation peut être effectuée en utilisant telle classe de fonctions que l'on sait facilement représenter. Il nous manque également une bonne implantation des séries rationnelles ; faut-il les représenter sous forme canonique ou se contenter de représentations non canoniques (par exemple par des expressions rationnelles ou des représentations linéaires non minimales) et déclancher des tests d'égalité (couteux) lorsque la question se pose?

Le point fort des séries génératrices tient au fait qu'elles sont intrinsèques. L'approximation de ces séries par des polynômes permet d'approximer localement le comportement Entrée/Sortie d'un système dynamique analytique. L'exemple donné en démonstration montre que le logiciel supporte facilement une approximation à l'ordre 7 d'une série génératrice en 3 variables. Une telle approximation fournit, pour un jeu d'entrées bornées, un développement de Taylor à l'ordre 7 de la sortie. L'algorithme a donc un intérêt pratique indéniable.

Le deuxième résultat est que l'on dispose maintenant d'une implantation en SCRATCHPAD des polynômes en variables non commutatives.

Cette thèse fait apparaitre qu'il y a quatre manières de les représenter:

1. La représentation distribuée creuse est utile pour effectuer des calculs d'algèbre linéaire (rang, noyau et image d'une application linéaire ...).
2. La représentation récursive creuse doit être considérée comme l'implantation de base. Elle est efficace pour les calculs de produits de Cauchy et les calculs de produits de mélange. Je compte l'utiliser pour calculer les bases-standard d'idéaux de polynômes.
3. La représentation dans la base de Poincaré-Birkoff-Witt est utile pour les calculs d'exponentielles de Lie et pour aborder certains calculs dans l'algèbre de Weyl définie par un ensemble de champs de vecteurs.
4. La représentation dans une base de l'algèbre de mélange est intéressante quand il s'agit d'éviter une explosion de la taille des données, explosion produite par le simple calcul du produit de mélange de deux polynômes.

L'implantation est terminée en ce qui concerne les trois premiers points. J'ai réalisé une implantation dans la base de Radford (voir 6.5.1) mais je n'en suis pas complètement satisfait. Il conviendrait donc de la refaire en choisissant cette fois la base duale de la base $PBWL$.

Le troisième résultat pratique concerne l'implantation efficace des polynômes de Lie dans la base de Lyndon et la maîtrise complète des exponentielles de Lie pour une algèbre nilpotente. Une telle maîtrise est une condition nécessaire (mais pas suffisante) pour aborder dans de bonnes conditions le probème du "Motion-planning".

Même si le compilateur de SCRATCHPAD présente de graves faiblesses, les idées qui soutendent ce système de calcul formel sont des idées d'avant-garde. Les concepteurs de SCRATCHPAD ont assimilé bon nombre de concepts-clef en matière de génie logiciel (programmation orientée objet, types abstraits de données, généricité). Pour caricaturer un peu, en SCRATCHPAD, on n'implante pas des algorithmes, **on construit des objets informatiques réutilisables.**

La programmation orientée objet est souvent utilisée pour écrire des logiciels de simulation. Supposons que l'on désire réaliser, par programme, la simulation du fonctionnement d'un métro et que l'on confie la réalisation de ce programme à dix personnes différentes. Il est fort probable que ces personnes modéliseront la réalité en "inventant" des classes d'objets informatiques tels que "vagons", "rames" et autres "stations" ... mais que la sémantique précise de ces objets sera différente d'une personne à l'autre. Bref, les objets du monde réel ne se laissent pas facilement *abstraire*.

S'agissant d'objets mathématiques, on bénéficie de tout un acquis (dû en particulier à l'école "Bourbaki") qui a su "inventer" et "hiérarchiser" des classes d'objets (structures) réutilisables et ... réutilisées. Le grand mérite de SCRATCHPAD est d'offrir un langage permettant de traduire assez naturellement tout ce savoir-faire des mathématiciens. Il s'ensuit que la programmation en SCRATCHPAD ne peut pas être à courte vue (pour un algorithme particulier) mais doit viser à offrir une implantation cohérente de la hiérarchie des grandes structures mathématiques.

Les autres systèmes de calcul formel, à ma connaissance, sont non typés ; ils sont plus ou moins basés sur un mécanisme universel de simplification d' expressions symboliques. Un tel simplificateur a le mérite d'être facile d'emploi en algèbre commutative. Il faut cependant s'attendre à des difficultés certaines quand il s'agit de manipuler une commutativité partielle telle qu'elle existe dans des polynômes non commutatifs dont les coefficients sont des polynômes en variables commutatives. L'introduction du typage en SCRATCHPAD lève toute difficulté.

L' algèbre non commutative est peu utilisée dans les systèmes de calcul formel ; certaines mauvaises langues diront qu'il en est ainsi parce qu'elle ne présente aucun intérêt. J'espère que cette thèse, axée sur des questions d'automatique non linéaire, contribuera à démontrer le contraire.

# Références


[1] J. Berstel and C. Reutenauer. *Les séries rationnelles et leurs langages.* Etudes et Recherches en Informatique. Masson, 1984.

[2] N. Bourbaki. *Groupes et Algèbres de Lie, Chapitre 2, algèbres de Lie libres.* Hermann, 1972.

[3] F. Boussemart. *La simulation graphique interactive des systèmes dynamiques non linéaires : conception et réalisation en Scratchpad.* Thèse de doctorat, Université Lille I, 24 Janvier (date prévue) 1992.

[4] B. Buchberger. *An algorithm for finding a basis for the residue class ring of zero-dimentional polynomial ideal.* Ph. d. thesis, Univ. of Innsbruck, Austria, Math. Inst., 25 Juin 1965.

[5] Nachumand Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems*, volume B of *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers, 1990.

[6] J. Dixmier. *Algèbres enveloppantes.* Paris : Bordas (Gauthier Villars), 1974.

[7] M. Fliess. Sur divers produits de séries formelles. *Bull. Soc. Math. Fr.*, 102:181–191, 1974.

[8] M. Fliess. Fonctionnelles causales non linéaires et indéterminées non commutatives. *Bull. Soc. Math. France*, 109:3–40, 1981.

[9] M. Fliess. Réalisation locale des systèmes non linéaires, algèbres de lie filtrées transitives et séries génératrices. *Invent. Math.*, 71:521–537, 1983.

[10] M.C. Gontard. Une première approche de la sémantique du langage de calcul formel scratchpad ii. Mémoire de dea, Université Paris 6, 1986.

[11] R. Hermann and A.J. Krener. Nonlinear controllability and observability. *IEEE Trans. Automat. Control*, 22:728–740, 1977.

[12] V. Hoang Ngoc Minh, G. Jacob, and N.E. Oussous. Input/output behaviour of non-linear analytic systems: rational approximations, nilpotent structural approximations. In J.P. Gauthier B. Bonnard, B. Birde and I. Kupka, editors, *Analysis of controlled dynamical systems*, pages 253–262, 1990.

[13] G. Jacob. Lyndon discretization and exact motion planning. In ??, editor, *European Control Conference*, pages 1507–1512, 1991.

[14] G. Jacob and N.E. Oussous. Sur un résultat de Ree : séries de lie et algèbres de mélange. Publication du LIFL IT–103, Université Lille I, 1987.

[15] N. Jacobson. *Lie Algebras*, volume 10 of *Interscience Tracts in Pure and Applied Mathematics*. Interscience Publisher, a division of Jhon Wiley & Sons, New York, London.

[16] B. Jakubczyk. Existence and uniqueness of realisations of nonlinear systems. *SIAM J. Control Optim.*, 18:455–471, 1980.

[17] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–376, 1970.

[18] P.V. Koseleff. Jeux de mots dans les algébres de lie libres : quelques bases et formules. *Theoret. Comput. Sci.*, 79(1):241–256, February 1991.

[19] G. Lafferrière and H.J. Sussmann. Motion planning for controllable systems without drift: a preliminary report. In J.P. Gauthier B. Bonnard, B. Birde and I. Kupka, editors, *Analysis of controlled dynamical systems*. Birkhauser, 1990.

[20] M. Lothaire. *Combinatorics on words*, volume 17 of *Encyclopedia of Mathematics and its applications*, chapter 5, pages 63–99. Addison-Wesley, Reading, Massachusetts, 1983.

[21] G. Melançon and C. Reutenauer. Lyndon words, free algebras and shuffles. Publication du LITP Paris et UQAM Montreal 87–63, Université Paris VII et Université du Quebec, 1987.

[22] F. Mora. Gröbner and standard bases for noncommutative polynomial rings. In *Private communication about a contribution to the conference AAECC-3, Grenoble*, 1985.

[23] F. Ollivier. *Le problème de l'identifiabilité structurelle globale : approche théorique, méthodes effectives et bornes de complexité*. Thèse de doctorat, Ecole Polytechnique, 25 Juin 1990.

[24] N.E. Oussous. *Etude et traitement des séries formelles non commutatives, pour la représentation minimale des systèmes dynamiques non linéaires*. Thèse de doctorat, Université Lille I, 14 Décembre 1988.

[25] N.E. Oussous. Computation, on macsyma, of the minimal differential representation of noncommutative polynomials. *Theoret. Comput. Sci.*, 79(1):241–256, February 1991.

[26] N.E. Oussous. Macsyma computation of local minimal realization of dynamical systems of which generating power series are finite. *J. Symbolic Computation*, 12:115–126, 1991.

[27] N.E. Oussous and M. Petitot. Polynômes non commutatifs : représentation et traitement par les systèmes de calcul formel. In G. Jacob M. Delest and P. Leroux, editors, *Séries formelles et combinatoire algébrique*, pages 349–368, 1991.

[28] D.E. Radford. A natural ring basis for the shuffle algebra and an application to group schemes. *Journal of Algebra*, 58:432–454, 1979.

[29] R. Ree. Lie elements and an algebra associated with shuffle. *Ann. of Math*, 68:210–220, 1958.

[30] C. Reutenauer. The local realisation of generating series of finite lie rank. In M. Fliess and M. Hazewinkel, editors, *Algebraic and Geometric Methods in Nonlinear Control Theory*, pages 33–43, 1986.

[31] A. Salomaa and M. Soittola. *Automata–theoretic aspects of formal power series.* Springer–Verlag, New York, 1978.

[32] H.J. Sussmann. Existence and uniqueness of minimal realisations of nonlinear systems. *Math. Systems Theory*, 10:263–284, 1977.

[33] H.J. Sussmann. Two new methods for motion planning for controllable systems without drift. In *European Control Conference*, pages 1501–1506, 1991.

[34] G. Viennot. *Algèbres de Lie libres et monoïdes libres*, volume 691 of *Lecture Notes In Mathematics.* Springer-Verlag, 1978.

# A

# A propos du typage

Nous allons essayer de mieux cerner l'originalité de SCRATCHPAD en le comparant à d'autres langages de programmation comme ADA ou EIFFEL. Nous verrons que la notion de type est une notion difficile à cerner et qu'elle répond selon les concepteurs à des préoccupations très diverses.

## A.1 Pourquoi des types ?

### A.1.1 Détecter le maximum d'erreurs à la compilation

Le système de typage *explicite* obligeant le programmeur à déclarer le type de chaque variable utilisée est dans ce cas précis, celui qui s'impose. Il s'agit d'exiger du programmeur un supplément d'information afin de mieux détecter les incohérences de son programme.

Nous avons vu au chapitre 2 que le typage en SCRATCHPAD est *implicite*:

> *Le type d'une variable peut être inféré par le compilateur lors de sa première affectation; il y a donc moins d'erreurs détectées à la compilation.*

Les contrôles de type sont effectués dans 2 cas précis:

1. affectation
2. passage de paramètre

La philosophie de ces contrôles est très variable d'un langage à l'autre mais le but est toujours le même: ne pas attendre l'exécution pour détecter certaines incohérences. Si l'on désire privilégier la sécurité, il faut interdire au compilateur d'effectuer des forçages de type non demandés explicitement.

### A.1.2 Rendre plus claire la sémantique du programme

Voici un petit exemple (en ADA) montrant l'intérêt des types *dérivés*:

```
type PRIX_DOLLARS is new FLOAT;
type PRIX_FRANCS  is new FLOAT;
```

```
pd :PRIX_DOLLARS;  pf:PRIX_FRANCS; s:FLOAT;
s:= pf+pd;                -- erreur detectee a la compilation
pd := s                   -- erreur
pd:=PRIX_DOLLARS(s);      -- correct
```

D'un point de vue purement opératoire, on peut parfaitement se passer des types dérivés **PRIX_DOLLARS** et **PRIX_FRANCS** mais cette information supplémentaire aide à rendre le programme beaucoup plus compréhensible. D'autre part, cela permet au compilateur de détecter certaines incohérences comme le fait d'additionner des **DOLLARS** avec des **FRANCS**.

### A.1.3 Générer automatiquement des contrôles à l'exécution

Exemple en Pascal:

```
var t:array[1..10] of INTEGER;
    i:INTEGER;
.....
t[i]:=0;  ....
```

Le compilateur génère du code chargé de contrôler à l'exécution que la variable $i$ est bien dans l'intervalle $[1, 10]$. Le mécanisme des *exceptions* en ADA permet au programmeur de faire un traitement de l'erreur en cas de débordement.

### A.1.4 Améliorer la rapidité d'exécution des programmes

#### A.1.4.1 Le cas PROLOG:

Le langage PROLOG dans sa conception originale n'est pas un langage typé. TURBO-PROLOG bien que tournant sur IBM-PC est beaucoup plus rapide que la plupart des autres PROLOG qui eux tournent sur des machines beaucoup plus puissantes.

La raison en est l'introduction du typage des données. Les prédicats ont alors leur signature qui est déclarée au début du programme. Ce supplément d'information, connu au moment de la compilation, est exploitée par le compilateur pour optimiser certains algorithmes.

#### A.1.4.2 La gestion de la mémoire:

Le typage *statique* d'une variable permet souvent de connaître la place qu'elle occupe en mémoire et donc de lui allouer une adresse dès la compilation.

Cette information suplémentaire, permet de gérer au mieux la mémoire, en évitant son morcellement. Le *Garbage Collector* très pénalisant quant au temps d'exécution des programmes n'est pas nécessaire. SCRATCHPAD ne fait pas grand cas de ce point de vue; les entiers représentés par un nombre fixe de bits sont peu utilisés. L'utilisation du type **Integer** qui permet de traiter des entiers arbitrairement grands est très commode mais se paie forcément en temps d'exécution.

### A.1.5 Simplifier la tâche de programmation

Des procédures de même nom peuvent correspondre à des traitements différents. L'information contenue dans le typage des paramètres sert alors à faire le lien entre le nom de la procédure invoquée et le traitement correspondant. Nous verrons que cette liaison peut être *statique* (faite à la compilation) ou *dynamique* (faite à l'exécution).

De toute façon, la possibilité de *surcharger* les opérateurs ou les fonctions apparaît comme indispensable en calcul formel. Ainsi un même nom d'opérateur + correspond à des traitements très différents suivant le type des opérandes (addition de 2 entiers, de 2 réels, de 2 matrices etc ...).

De tous les services que l'on peut attendre du typage des objets, SCRATCHPAD a privilégié ce dernier point de vue, à savoir l'utilisation de la *surcharge*.

## A.2 Typage statique et typage dynamique

Rappelons que le typage d'un objet induit les informations suivantes sur cet objet:

1. Sa représentation en mémoire
2. L'ensemble de ses valeurs possibles
3. L'ensemble des opérations permises sur cet objet
4. Un moyen de faire le lien entre le nom d'une opération permise et le traitement correspondant à effectuer

A travers cet essai de définition, nous percevons toute l'ambiguité possible de la notion de type.

### A.2.1 Typage statique

Il y a typage statique lorsque le type des objets est connu à la compilation. Cette solution est avantageuse lorsque l'on veut privilègier la sécurité et l'efficacité des programmes.

C'est le choix fait par les concepteurs du langage ADA. Le compilateur est capable de parfaitement déterminer les traitements à effectuer en se basant sur le nom des procédures et sur le type de leurs paramètres.

Ce choix n'exclut pas que certains contrôles soient faits à l'exécution et ne puissent en aucun cas être effectués avant. C'est le cas de tous les contrôles de non-débordement:

```
subtype NATURAL is INTEGER range 0..INTEGER 'LAST;
i:INTEGER; n:NATURAL;
......
n:=i;
```

Il est impossible de déterminer à la compilation si l'affectation `n:=i` est correcte car la valeur de la variable $i$ n'est pas connue. Ceci prouve la pertinance du concept de *sous-type* en ADA.

Un sous-type n'est pas un nouveau type:

- les opérations permises ainsi que la représentation en mémoire sont les mêmes.
- les affectations entre sous-types d'un même type sont possibles sans *forçage de type.*
- pour des sous-types d'un même type, deux appels de nom identique déclanchent des traitements rigoureusement identiques.

La notion de sous-type est une notion fondamentalement dynamique. Le sous-type d'un objet caractérise l'ensemble de ses valeurs possibles; cette information n'étant utile qu'en phase d'exécution, il parfaitement logique d'autoriser la création dynamique de sous-types. Ainsi

```
-- i et j sont des variables de type INTEGER
subtype ENTIER_LOCAL is INTEGER range i..j;
```

est une déclaration admise quoique les valeurs des variables $i$ et $j$ ne soient pas connues par le compilateur.

Malheureusement en SCRATCHPAD le concept de *SubDomain* est beaucoup moins clair.
**Exemple:**

```
NonNegativeInteger == SubDomain(Integer,#1 >= 0) add
   x,y:$
   sup(x,y) == MAX(x,y)$Lisp
   x - y ==
     (y:Integer >$Integer x:Integer) => "failed"
     (x:Integer -$Integer y:Integer):$
```

Nous constatons que le traitement correspondant à une soustraction a été réécrit et que sa signature en est modifiée.

Les domaines `Integer` et `NonNegativeInteger` n'exportent pas les mêmes opérations quoique l'un soit un *sous-domaine* de l'autre:

```
(1) ->)sh I
 Integer is a domain constructor.
 Abbreviation for Integer is I
 This constructor is currently exposed.
 Issue )edit /spad/nalgebra/basicd2.spad to see algebra source code for I

------------------------------- Operations -------------------------------
 ......
 -@ : $ -> $                          @-@ : ($,$) -> $
 SqFr : $ -> FF $                     abs : $ -> $
 abs : $ -> $                         oddp : $ -> B
 associates? : ($,$) -> B             characteristic : () -> NNI
```

```
deriv : $ -> $                          factor : $ -> FF $
numberOfDigits : ($,$) -> $
......


(1) ->)sh NNI
 NonNegativeInteger is a domain constructor.
 Abbreviation for NonNegativeInteger is NNI
 This constructor is currently exposed.
 Issue )edit /spad/nalgebra/basicd2.spad to see algebra source code for NNI

------------------------------- Operations -------------------------------

 @*@ : ($,$) -> $                       @*@ : (NNI,$) -> $
 @**@ : ($,NNI) -> $                    @+@ : ($,$) -> $
 @-@ : ($,$) -> Union($,"failed")       @<@ : ($,$) -> B
 @=@ : ($,$) -> B                       1 : () -> $
 0 : () -> $                            coerce : $ -> E
 -- coerce : E -> Union($,"failed")     gcd : ($,$) -> $
 max : ($,$) -> $                       min : ($,$) -> $
 @quo@ : ($,$) -> $                     @rem@ : ($,$) -> $
 sup : ($,$) -> $
 @div@ : ($,$) -> Record(quotient: $,remainder: $)
 @exquo@ : ($,$) -> Union($,"failed")
```

Tout cela est une source de difficulté pour les débutants en SCRATCHPAD.

Notons toutefois que la question de l'articulation de la notion de *sous-domaine* et de *sous-catégorie* est délicate; il faut distinguer deux notions:

1. La où une valeur appartenant au domaine `I` est requise, une valeur appartenant au domaine `NNI` convient.

2. La où le domaine `I` convient, le domaine `NNI` ne convient pas forcément car `I` est un anneau tandis que `NNI` n'est qu'un semi-anneau.

La distinction est mal traitée dans la plupart des langages de programmation et malheureusement SCRATCHPADne fait pas exception.

### A.2.2 Typage dynamique

> *Il y a typage dynamique lorsque le type des objets peut varier en cours d'exécution du programme.*

Cette solution est intéressante lorsque l'on désire utiliser le mécanisme de la *liaison dynamique* illustré par l'exemple suivant [1]:

[1] rédigé dans un langage imaginaire

```
-- declaration de variables
      f:FIGURE   ;
      t:TRIANGLE ;
      c:CERCLE   ;
-- procedures
      procedure tracer(x:TRIANGLE) is ....;
      procedure tracer(x:CERCLE)   is ....;
-- traitement
      .....
      if ( ... ) then f:=t  else f:=c ;
      tracer(f) ;
```

Il est évident que `tracer(f)` doit se traduire différemment selon que le type dynamique de *f* sera `TRIANGLE` ou `CERCLE`. Le choix définitif du traitement à effectuer ne peut pas être fait à la compilation car à cet instant, le type *dynamique* de l'objet *f* n'est pas connu.

Ce genre de programme peut être réécrit en utilisant un typage statique mais il faut, dans ce cas, utiliser des types avec *discriminant* et définir le type `FIGURE` comme suit [2]:

```
FIGURE == union(TRIANGLE,CERCLE);
tracer(f: FIGURE) ==
   f case TRIANGLE => tracer1triangle(f::TRIANGLE)
   f case CERCLE => tracer1cercle(f::CERCLE)
traitement ==
   f:FIGURE ;  t:TRIANGLE  ; c:CERCLE
   ....
   if ( ... ) then f:=t  else f:=c
   tracer(f) ;
```

Nous constatons, sur cet exemple, que le typage statique oblige le programmeur à utiliser une structure de contrôle du type *case*.

Le choix d'un typage dynamique implique que l'information de typage apparaisse dans la représentation interne des objets en mémoire. C'est en effet le seul moyen de garder une trace des changements de type d'un objet, pendant l'exécution du programme.

### A.2.3 Un compromis possible

Le choix d'un typage *complètement* dynamique a le grave inconvénient de reporter la détection des erreurs uniquement en phase d'exécution. Une solution bien meilleure est fournie dans le langage EIFFEL qui essaie d'allier les avantages des deux méthodes de typage à savoir:

- Sécurité du typage statique
- Souplesse du typage dynamique

L'idée de base est à peu près la suivante: les *classes* `TRIANGLE` et `CERCLE` vont hériter de la classe `FIGURE`. Les variables sont déclarées, ce qui leur confère un type statique; ainsi la variable *f* aura pour type statique `FIGURE`.

[2] syntaxe à la SCRATCHPAD

Le type dynamique des variables peut varier en cours d'exécution du programme mais ce *polymorphisme* est contrôlé par les liens d'héritage, ce qui permet de détecter la majorité des erreurs à la compilation [3]. Ainsi le type dynamique de *f* de `FIGURE` peut devenir `TRIANGLE` ou `CERCLE`.

Le compilateur EIFFEL impose que le type d'une variable évolue du général au particulier, ainsi après les déclarations

```
f:FIGURE   ; t:TRIANGLE ; c:CERCLE   ;
```

les affectations

```
f:=t ; f:=c;
```

sont correctes tandis que les affectations

```
t:=f ; c:=f;
```

provoqueront une erreur à la compilation.

## A.3 Conclusion

Le but de cette annexe était de montrer que la notion de type n'est pas simple; de fait, c'est encore un problème ouvert qui intéresse tous les chercheurs en génie logiciel.

Cette étude permettra au lecteur de situer le système de typage de SCRATCHPAD. Celui-ci est **statique** et **implicite**. Le point fort de SCRATCHPAD est avant tout l'usage massif qu'il fait de la généricité. Cette généricité est contrôlée par la notion de **catégorie**, notion qui manque beaucoup en ADA.

Dans ce chapitre, nous avons délibèrement ignoré une autre piste de recherche possible en matière de *type abstrait*. Il s'agit de la possibilité, pour un logiciel, de générer du code exécutable uniquement à partir de la *spécification* d'un type abstrait. Un exemple très classique est fourni par l'axiomatique de PEANO définissant le type `Integer`:

```
-- signatures des fonctions
      0   : () -> Integer
      s   : Integer -> Integer        -- fonction successeur
     "+"  : (Integer,Integer) -> Integer
     "*"  : (Integer,Integer) -> Integer
-- regles de reecriture
      0  + x  -> x
    s(x) + y  -> s(x+y)
      0  * x  -> 0
    s(x) * y  -> y + x*y
```

Il est alors possible de déduire automatiquement [4] à partir de ce système de réécriture, la

[3] EIFFEL n'a pas réussi jusqu'à présent à règler de façon complètement satisfaisante les contradictions possibles entre les 2 systèmes de typage

[4] Formes normales d'un système confluent et noethérien

représentation interne des **Integer** à savoir $s(s...s(0)) = s^n(0)$; ce genre d'idée est exploitée par le langage OBJ mais il est assez évident que la représentation interne induite dans ce cas précis, n'est pas aussi bonne [5] que la représentation habituelle en base 2.

[5] place mémoire et temps calcul prohibitifs

# B

# Calcul du produit de mélange

## B.1 Le fichier source

Voici le contenu d'un fichier permettant de tester l'efficacité des représentations récursives et distribuées pour les polynômes en variables non commutatives.

```
-- fichier de test  XRPOLY : polynomes non commutatifs (recursifs)
--                  XDPOLY : polynomes non commutatifs (distribues)
-- comparaison des resultats (calculs de shuffle)
-----------------------------------------------------------------------
)load XRPOLY XDPOLY )cond
)time on
)clear all

alph := OV [z,y,x]
mot  := OFMON1 alph
rpoly:= XRPOLY(alph,I)
dpoly:= XDPOLY(alph,I)

x:alph:=x
y:alph:=y
z:alph:=z
-----------------------------------------------------------------------
pr: rpoly := 2*x+3*y+5
pr2 := pr*pr

pd: dpoly := extend(pr)
pd2 :=pd*pd

extend(pr2) =$dpoly pd2

pr6: rpoly := pr2*pr2*pr2
pd6: dpoly := pd2*pd2*pd2
extend(pr6) - pd6

qr := sh(pr6,pr2)
```

```
qd := sh(pd6,pd2)
extend(qr)-qd
```

## B.2 Trace d'exécution

On constate la supériorité de la représentation récursive pour les calculs de "shuffle" et de produit (de Cauchy).

```
;alph := OV [z,y,x]

   (1)  OrderedVarlist [z,y,x]
                                                        Type: Domain
                                  Time: .333 (IN) + 1.867 (OT) = 2.2 sec

;mot  := OFMON1 alph

   (2)  OrderedFreeMonoid1 OrderedVarlist [z,y,x]
                                                        Type: Domain
                                                Time: .1 (IN) = .1 sec

;rpoly:= XRPOLY(alph,I)

   (3)  XRecursivePolynomial(OrderedVarlist [z,y,x],Integer)
                                                        Type: Domain
                                                Time: .1 (IN) = .1 sec

;dpoly:= XDPOLY(alph,I)

   (4)  XDistribuedPolynomial(OrderedVarlist [z,y,x],Integer)
                                                        Type: Domain
                                              Time: .067 (IN) = .067 sec

;x:alph:=x

   (5)  x
                                            Type: OrderedVarlist [z,y,x]
                                  Time: .3 (IN) + .533 (OT) = .833 sec

;y:alph:=y

   (6)  y
                                            Type: OrderedVarlist [z,y,x]
                                                  Time: .1 (IN) = .1 sec

;z:alph:=z

   (7)  z
```

```
                                                    Type: OrderedVarlist [z,y,x]
                                                                  Time: 0 sec

;pr: rpoly := 2*x+3*y+5

   (8)  5 + x 2 + y 3
                Type: XRecursivePolynomial(OrderedVarlist [z,y,x],Integer)
                       Time: 2.067 (IN) + .2 (EV) + 2.167 (OT) = 4.433 sec

;pr2 := pr*pr

   (9)  25 + x(20 + x 4 + y 6) + y(30 + x 6 + y 9)
                Type: XRecursivePolynomial(OrderedVarlist [z,y,x],Integer)
                                    Time: .267 (IN) + .067 (OT) = .333 sec

;pd: dpoly := extend(pr)

   (10)  5 + 2x + 3y
                Type: XDistribuedPolynomial(OrderedVarlist [z,y,x],Integer)
                              Time: .3 (IN) + .3 (EV) + .133 (OT) = .733 sec

;pd2 :=pd*pd

                                  2                    2
   (11)  25 + 20x + 30y + 4x  + 6x y + 6y x + 9y
                Type: XDistribuedPolynomial(OrderedVarlist [z,y,x],Integer)
                                       Time: .2 (IN) + .233 (EV) = .433 sec

;extend(pr2) =$dpoly pd2

   (12)  true
                                                                  Type: Boolean
                                 Time: .4 (IN) + .1 (EV) + .8 (OT) = 1.3 sec

;pr6: rpoly := pr2*pr2*pr2

   (13)  ...
                Type: XRecursivePolynomial(OrderedVarlist [z,y,x],Integer)
                              Time: .533 (IN) + .067 (EV) + .1 (OT) = .7 sec

;pd6: dpoly := pd2*pd2*pd2

   (14)
                                          2                                  2
     15625 + 37500x + 56250y + 37500x  + 56250x y + 56250y x + 84375y
   +
           3          2                              2           2
     20000x  + 30000x y + 30000x y x + 45000x y  + 30000y x  + 45000y x y
   +
```

```
          2          3        4        3         2           2 2
    45000y x + 67500y  + 6000x  + 9000x y + 9000x y x + 13500x y
  +

  ..... resultat sur 70 lignes

               2          2 2          2             3           4
    324y x y x y  + 216y x y x  + 324y x y x y + 324y x y x + 486y x y
  +
        2 4       2 3        2 2          2 2 2       2     2
    144y x  + 216y x y + 216y x y x + 324y x y  + 216y x y x
  +
        2             2   2        2   3       3 3       3 2
    324y x y x y + 324y x y x + 486y x y  + 216y x  + 324y x y
  +
        3           3   2       4 2       4          5       6
    324y x y x + 486y x y  + 324y x  + 486y x y + 486y x + 729y
          Type: XDistribuedPolynomial(OrderedVarlist [z,y,x],Integer)
                   Time: .833 (IN) + 1.4 (EV) + .433 (OT) = 2.667 sec

;extend(pr6) - pd6

   (15)  0
          Type: XDistribuedPolynomial(OrderedVarlist [z,y,x],Integer)
                   Time: .067 (IN) + .267 (EV) + .133 (OT) = .467 sec

;qr := sh(pr6,pr2)

   (16)   .......
           Type: XRecursivePolynomial(OrderedVarlist [z,y,x],Integer)
                      Time: .1 (IN) + 19.5 (EV) + .1 (OT) = 19.7 sec

;qd := sh(pd6,pd2)

   (17)
                                              2
    390625 + 1250000x + 1875000y + 2500000x  + 3750000x y + 3750000y x
  +
             2           3           2                                2
    5625000y  + 3200000x  + 4800000x y + 4800000x y x + 7200000x y
  +
              2                          2            3           4
    4800000y x  + 7200000y x y + 7200000y x + 10800000y  + 2650000x
  +
           ' 3            2              2 2              2
    3975000x y + 3975000x y x + 5962500x y  + 3975000x y x
  +

   ..... resultat sur 380 lignes ......
```

```
        4 4          4 3           4 2             4 2 2          4     2
  36288y x  + 54432y x y + 54432y x y x + 81648y x y  + 54432y x y x
+
        4                4   2           4   3          5 3
  81648y x y x y + 81648y x y x + 122472y x y  + 54432y x
+
        5 2          5                5   2          6 2           6
  81648y x y + 81648y x y x + 122472y x y  + 81648y x  + 122472y x y
+
         7           8
  122472y x + 183708y
        Type: XDistribuedPolynomial(OrderedVarlist [z,y,x],Integer)
              Time: .1 (IN) + 117.6 (EV) + .133 (OT) = 117.833 sec

;extend(qr)-qd

  (18)  0
        Type: XDistribuedPolynomial(OrderedVarlist [z,y,x],Integer)
                          Time: 10.833 (EV) + .267 (OT) = 11.1 sec
```

# C

# Polynômes de Lie

## C.1 Fichier source

Le fait de disposer de plusieurs implantations d'une même structure mathématique permet de facilement vérifier la correction des programmes, ce qui est toujours un problème surtout lorsque l'on effectue de la maintenance logicielle.

Voici un petit fichier de test des polynômes de Lie où les mêmes calculs sont systèmatiquement effectués deux fois (dans la base de Lyndon et dans l'algèbre enveloppante).

```
-- Test du domaine LPOLY  polynomes de Lie
-- V4.3

)load XFALG LALG XDPOLY FLALG MAGMA LWORD LPOLY LPOLY2 )cond
)time on
)clear all

lettre  := OV1 [a,b]
lpoly   := LiePolynomial(lettre,RN)
lpoly2  := LiePolynomial2(lettre,RN)

DPOLY := XDistribuedPolynomial(lettre,RN)
LW    := LyndonWord lettre

a:lettre := variable('a)
b:lettre := variable('b)

liste : List LW := LyndonWordsList([a,b], 5)

s  :lpoly  := LiePoly(liste.4) - LiePoly(liste.8)
s2 :lpoly2 := LiePoly(liste.4) - LiePoly(liste.8)

t :lpoly  := s  + 2 *$lpoly  LiePoly(liste.3) - 5 *$lpoly  LiePoly(liste.5)
t2:lpoly2 := s2 + 2 *$lpoly2 LiePoly(liste.3) - 5 *$lpoly2 LiePoly(liste.5)

d :DPOLY := t2 :: DPOLY;  -- conversion poly. distribue -> poly. de Lie
LiePolyIfCan(d)$lpoly

t  := mirror t
```

```
t2 := mirror t2
t :: DPOLY - t2:: DPOLY

t  := [s,t]
t2 := [s2,t2]
t ::DPOLY - t2 :: DPOLY
```

## C.2 Trace d'exécution

On remarquera que l'implantation sous forme développée des polynômes de Lie est assez gourmande en place mémoire.

```
;lettre  := OV1 [a,b]

   (1)  OrderedVarlist1 [a,b]
                                                        Type: Domain
                                                Time: .1 (OT) = .1 sec

;lpoly   := LiePolynomial(lettre,RN)

   (2)  LiePolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                                        Type: Domain
                                                         Time: 0 sec

;lpoly2  := LiePolynomial2(lettre,RN)

   (3)  LiePolynomial2(OrderedVarlist1 [a,b],RationalNumber)
                                                        Type: Domain
                                        Time: .1 (IN) + .4 (OT) = .5 sec

;DPOLY := XDistribuedPolynomial(lettre,RN)

   (4)  XDistribuedPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                                        Type: Domain
                                            Time: .067 (IN) = .067 sec

;LW     := LyndonWord lettre

   (5)  LyndonWord OrderedVarlist1 [a,b]
                                                        Type: Domain
                                    Time: .1 (IN) + .067 (OT) = .167 sec

;a:lettre := variable('a)

   (6)  a
                                                Type: OrderedVarlist1 [a,b]
```

```
                                        Time: .1 (IN) + .1 (OT) = .2 sec

;b:lettre := variable('b)

   (7)  b
                                              Type: OrderedVarlist1 [a,b]
                                        Time: .6 (EV) + .1 (OT) = .7 sec

;liste : List LW := LyndonWordsList([a,b], 5)

   (8)
                      2       2     3      2 2       3     4
   [{a}, {b}, {a b}, {a b}, {a b }, {a b}, {a b }, {a b }, {a b},

     3 2     2          2 3          2       4
    {a b }, {a b a b}, {a b }, {a b a b }, {a b }]

                            Type: List LyndonWord OrderedVarlist1 [a,b]
                                  Time: .067 (IN) + .1 (OT) = .167 sec

;s  :lpoly  := LiePoly(liste.4) - LiePoly(liste.8)

           2      4
   (9)  {a b } - {a b}
            Type: LiePolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                  Time: .667 (IN) + .1 (EV) = .767 sec

;s2 :lpoly2 := LiePoly(liste.4) - LiePoly(liste.8)

           2             2     4      3        2  2         3      4
   (10)  a b  - 2b a b + b a - a b + 4a b a - 6a b a  + 4a b a  - b a
            Type: LiePolynomial2(OrderedVarlist1 [a,b],RationalNumber)
                        Time: .8 (IN) + .667 (EV) + .033 (OT) = 1.5 sec

;t :lpoly  := s  + 2 *$lpoly  LiePoly(liste.3) - 5 *$lpoly  LiePoly(liste.5)

           2         2       3       4
   (11)  2{a b} + {a b } - 5{a b} - {a b}
            Type: LiePolynomial(OrderedVarlist1 [a,b],RationalNumber)
                       Time: 2.167 (IN) + .767 (EV) + .267 (OT) = 3.2 sec

;t2:lpoly2 := s2 + 2 *$lpoly2 LiePoly(liste.3) - 5 *$lpoly2 LiePoly(liste.5)

   (12)
      2                2       2              2      3       2
     2a b - 4a b a + a b  + 2b a  - 2b a b + b a - 5a b + 15a b a
   +
              2       3    4      3        2  2         3      4
     - 15a b a  + 5b a  - a b + 4a b a - 6a b a  + 4a b a  - b a
```

```
          Type: LiePolynomial2(OrderedVarlist1 [a,b],RationalNumber)
                 Time: 1.667 (IN) + .167 (EV) + .167 (OT) = 2.0 sec

;d :DPOLY := t2 :: DPOLY;  -- conversion poly. distribue -> poly. de Lie
                                                              Type: Void
                                              Time: .167 (IN) = .167 sec

;LiePolyIfCan(d)$lpoly

           2         2       3       4
   (14)  2{a b} + {a b } - 5{a b} - {a b}
Type: Union(LiePolynomial(OrderedVarlist1 [a,b],RationalNumber),"failed")
                                Time: .167 (EV) + .033 (OT) = .2 sec

;t  := mirror t

           2         2       3       4
   (15)  2{a b} + {a b } + 5{a b} - {a b}
           Type: LiePolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                             Time: .1 (OT) = .1 sec

;t2 := mirror t2

   (16)
      2                2       2             2      3       2
     2a b - 4a b a + a b  + 2b a  - 2b a b + b a + 5a b - 15a b a
   +
           2       3    4      3        2  2         3      4
     15a b a  - 5b a  - a b + 4a b a - 6a b a  + 4a b a  - b a
           Type: LiePolynomial2(OrderedVarlist1 [a,b],RationalNumber)
                                Time: .067 (EV) + .067 (OT) = .133 sec

;t :: DPOLY - t2:: DPOLY

   (17)  0
     Type: XDistribuedPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                Time: .233 (IN) + .1 (OT) = .333 sec

;t  := [s,t]

   (18)
         2     2       2 2          3     2       3 2          2   2 2
     - 2{a b a b } - 2{a b a b} - 5{a b a b } - 5{a b a b} + 5{a b a b }
   +
         4   2         4   3
     - 2{a b a b} - 5{a b a b}
           Type: LiePolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                Time: .067 (IN) + .2 (EV) = .267 sec
```

```
;t2 := [s2,t2]

   (19)
        2     2     2 2        2 3          2 2
     - 2a b a b  + 4a b a b - 2a b a + 4a b a b  - 8a b a b a b
   +
           2        2 2        2            3 2       3 2       2
     4a b a b a + 2a b a b - 4a b a b a + 2a b a  - 2b a b  + 4b a b a b
   +
         2 2            2                            2 2     2 3
     - 2b a b a - 4b a b a b + 8b a b a b a - 4b a b a  + 2b a b
   +
      ... resultat sur une page
   +
       4   2          4       2     4 2 3      3   4       3   3
     15a b a b a - 15a b a b a  + 5a b a  + 25a b a b - 80a b a b a
   +
       3   2   2      3       3     3 2 4      2   5       2   4
     90a b a b a  - 40a b a b a  + 5a b a  - 45a b a b + 150a b a b a
   +
         2   3   2      2   2   3      2       4          6
     - 180a b a b a  + 90a b a b a  - 15a b a b a  + 35a b a b
   +
             5              4   2          3   3          2   4
     - 120a b a b a + 150a b a b a  - 80a b a b a  + 15a b a b a
   +
           7         6           5   2        4   3       3   4
     - 10b a b + 35b a b a - 45b a b a  + 25b a b a  - 5b a b a
           Type: LiePolynomial2(OrderedVarlist1 [a,b],RationalNumber)
                                Time: .8 (EV) + .067 (OT) = .867 sec

;t ::DPOLY - t2 :: DPOLY

   (20)  0
     Type: XDistribuedPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                    Time: 11.5 (IN) + .1 (EV) + .017 (OT) = 11.617 sec
```

# D

# La base PBWL

## D.1 Le fichier de commandes

Voici un petit fichier permettant d'effectuer des calculs dans la base $PBWL$ et de s'auto-vérifier.

```
-- Test XPBWPOLY       Polynomes non comutatifs exprimes dans la base
--                     de Poincare-Birkoff-Witt
-- V4.3 du 26/07/91
------------------------------------------------------------------------
)load OV1 OFMON1 LWORD XDPOLY PBWLB XPBWPOLY )cond
)clear all

LETTRE := OV1 [a,b]
a:LETTRE := variable('a)
b:LETTRE := variable('b)

MOT   := OFMON1 LETTRE
LW    := LWORD(LETTRE)
BASE  := PBWLB LETTRE
LP    := LPOLY(LETTRE, RN)
POLY  := XPBWPOLY(LETTRE, RN)
DPOLY := XDPOLY(LETTRE, RN)

liste : List LW := LyndonWordsList([a,b], 6)

p : POLY := monom(liste.10 :: BASE, 1)
q : POLY := monom(liste.22 :: BASE, 1)
r : POLY := p*q

pd: DPOLY := p
qd: DPOLY := q
rd: DPOLY := p*q
rd - r::DPOLY

p*q - q*p
```

## D.2 Les résultats

Les crochets de Lyndon sont notés $\{w\}$ où $w$ est un mot de Lyndon. Ainsi $\{ab\}$ doit être interprété comme le polynôme de Lie $ab - ba$.

On remarquera (voir dernière ligne) qu'il est trés facile de vérifier si un polynôme exprimé dans la base $PBWL$ est un polynôme de Lie.

```
;LETTRE := OV1 [a,b]

   (1)  OrderedVarlist1 [a,b]
                                                          Type: Domain
                                                Time: .1 (IN) = .1 sec

;a:LETTRE := variable('a)

   (2)  a
                                                Type: OrderedVarlist1 [a,b]
                                      Time: .067 (IN) + .1 (OT) = .167 sec

;b:LETTRE := variable('b)

   (3)  b
                                                Type: OrderedVarlist1 [a,b]
                                      Time: .033 (IN) + .1 (OT) = .133 sec

;MOT   := OFMON1 LETTRE

   (4)  OrderedFreeMonoid1 OrderedVarlist1 [a,b]
                                                          Type: Domain
                                                Time: .067 (IN) = .067 sec

;LW    := LWORD(LETTRE)

   (5)  LyndonWord OrderedVarlist1 [a,b]
                                                          Type: Domain
                                                Time: .067 (OT) = .067 sec

;BASE  := PBWLB LETTRE

   (6)  PoincareBirkoffWittLyndonBasis OrderedVarlist1 [a,b]
                                                          Type: Domain
                                                Time: .1 (OT) = .1 sec

;LP    := LPOLY(LETTRE, RN)

   (7)  LiePolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                                          Type: Domain
                                                           Time: 0 sec
```

```
;POLY  := XPBWPOLY(LETTRE, RN)

   (8)  XPBWPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                                        Type: Domain
                                                Time: .067 (IN) = .067 sec

;DPOLY := XDPOLY(LETTRE, RN)

   (9)  XDistribuedPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                                        Type: Domain
                                                Time: .1 (IN) = .1 sec

;liste : List LW := LyndonWordsList([a,b], 6)

   (10)
                        2       2     3      2 2       3     4
   [{a}, {b}, {a b}, {a b}, {a b }, {a b}, {a b }, {a b }, {a b},

     3 2     2          2 3           2       4     5      4 2
    {a b }, {a b a b}, {a b }, {a b a b }, {a b }, {a b}, {a b },

     3          3 3     2     2     2 2        2 4          3
    {a b a b}, {a b }, {a b a b }, {a b a b}, {a b }, {a b a b },

        5
    {a b }]

                              Type: List LyndonWord OrderedVarlist1 [a,b]
                                                Time: .1 (OT) = .1 sec

;p : POLY := monom(liste.10 :: BASE, 1)

           2
   (11)  {a b a b}
           Type: XPBWPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                Time: .733 (IN) + .1 (EV) = .833 sec

;q : POLY := monom(liste.22 :: BASE, 1)

            5
   (12)  {a b }
           Type: XPBWPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                Time: .5 (IN) + .067 (OT) = .567 sec

;r : POLY := p*q

   (13)
     2         5       2     5         2 5                5   2
   {a b a b a b } + 2{a b a b a b} + {a b a b a b} + {a b }{a b a b}
```

```
          Type: XPBWPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                  Time: .333 (IN) + .1 (EV) = .433 sec

;pd: DPOLY := p

   (14)
      2         2 2          2                    2 2       3        2
     a b a b - a b a - 3a b a b + 4a b a b a - a b a  + 2b a b - 3b a b a
   +
           2
     b a b a
     Type: XDistribuedPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                      Time: .1 (IN) + .1 (OT) = .2 sec


;qd: DPOLY := q

            5         4      2   3      3   2     4       5
   (15)  a b  - 5b a b  + 10b a b  - 10b a b  + 5b a b - b a
     Type: XDistribuedPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                         Time: .133 (IN) = .133 sec

;rd: DPOLY := p*q

   (16)
      2         5     2     2   4      2     3   3      2     4   2
     a b a b a b  - 5a b a b a b  + 10a b a b a b  - 10a b a b a b
   +
       2     5       2     6     2 2 2 5     2 2       4      2 2   2   3
     5a b a b a b - a b a b a - a b a b  + 5a b a b a b  - 10a b a b a b
   +
        2 2   3   2     2 2   4       2 2   5          2     5
     10a b a b a b  - 5a b a b a b + a b a b a - 3a b a b a b
   +
           2 2   4          2 3   3          2 4   2          2 5
     15a b a b a b  - 30a b a b a b  + 30a b a b a b  - 15a b a b a b
   +
          2 6              2 5                    4                2   3
     3a b a b a + 4a b a b a b  - 20a b a b a b a b  + 40a b a b a b a b
   +
                  3   2               4                  5       2 3 5
     - 40a b a b a b a b  + 20a b a b a b a b - 4a b a b a b a - a b a b
   +
        2 2     4        2 2 2   3        2 2 3   2       2 2 4
     5a b a b a b  - 10a b a b a b  + 10a b a b a b  - 5a b a b a b
   +
       2 2 5        3     5        3 2   4        3 3   3        3 4   2
     a b a b a + 2b a b a b  - 10b a b a b  + 20b a b a b  - 20b a b a b
   +
```

```
        3 5          3 6        2   2 5        2         4
   10b a b a b - 2b a b a - 3b a b a b  + 15b a b a b a b
  +
          2     2   3        2     3   2        2     4
   - 30b a b a b a b  + 30b a b a b a b  - 15b a b a b a b
  +
       2     5          3 5          2     4           2 2   3
   3b a b a b a + b a b a b  - 5b a b a b a b  + 10b a b a b a b
  +
              2 3   2          2 4            2 5
   - 10b a b a b a b  + 5b a b a b a b - b a b a b a
     Type: XDistribuedPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                    Time: .3 (IN) + .167 (EV) + .067 (OT) = .533 sec

;rd - r::DPOLY

   (17)  0
     Type: XDistribuedPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                   Time: 3.8 (IN) + .067 (EV) + .067 (OT) = 3.933 sec

;p*q - q*p

           2         5       2     5         2 5
   (18)  {a b a b a b } + 2{a b a b a b} + {a b a b a b}
           Type: XPBWPolynomial(OrderedVarlist1 [a,b],RationalNumber)
                                Time: .467 (IN) + .1 (EV) = .567 sec
```

# E

# La réalisation minimale

## E.1 Le fichier source

Voici le contenu d'un fichier permettant de lancer sous interprèteur SCRATCHPAD le calcul de la réalisation minimale d'un polynôme de degré 7 en 3 variables.

```
-- test de DPS
-- V4.3 Aout 91
-- =============
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++ But: a partir d'un polynome g en variables non commutatives,
++      calculer un systeme dynamique analytique minimal admettant
++      g comme serie generatrice.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

)clear all
)load OFMON1 LWORD XRPOLY XDPOLY LPOLY VSBASIS1 IV )cond
)load DSMOD VECFIELD XTAYLOR1 DPS )cond

ca := IndexedVariable(x);                    -- Alphabet de commande
x0 :ca := var(0);
x1 :ca := var(1);
x2 :ca := var(2);

poly    := XRecursivePolynomial(ca,RN)  -- instanciation des domaines
sdyn    := DynamicPolynomialSystem(ca,'q,RN)

g:poly := 4*x2**4 - x0*x1**4 +3*x2*x1**3;
g := sh(g,x0);
g := sh(g,x1);
extend g

sd:sdyn := realisation(g);      -- algo de realisation

dimension sd            -- dimension de l'espace d'etat
Observation sd          -- fonction d'observation
VectField(sd,x0)        -- champ de vecteurs obtenu pour chaque entree
VectField(sd,x1)
VectField(sd,x2)
```

## E.2 Trace d'exécution

On constate sur cet exemple que le calcul complet s'effectue en moins d'une minute sur un PC-RT (diviser le temps par 3 pour une RS-6000).

```
;poly    := XRecursivePolynomial(ca,RN)  -- instanciation des domaines

   (5)  XRecursivePolynomial(IndexedVariable x,RationalNumber)
                                                          Type: Domain
                                                Time: .1 (IN) = .1 sec

;sdyn    := DynamicPolynomialSystem(ca,'q,RN)

   (6)  DynamicPolynomialSystem(IndexedVariable x,q,RationalNumber)
                                                          Type: Domain
                                              Time: .033 (IN) = .033 sec

;g:poly := 4*x2**4 - x0*x1**4 +3*x2*x1**3;
                                                          Type: Void
                                    Time: 1.667 (IN) + .233 (EV) = 1.9 sec

;g := sh(g,x0);
                                                          Type: Void
                                      Time: .1 (IN) + .1 (EV) = .2 sec

;g := sh(g,x1);
                                                          Type: Void
                                      Time: .1 (IN) + .2 (EV) = .3 sec

;extend g

   (10)
            3          4          4          3        2  2
     3x x x x   + 4x x x   + 12x x x   + 4x x x x   + 4x x  x x
       0 1 2 1      0 1 2      0 2 1      0 2 1 2      0 2  1 2
   +
         3          4          3          4          3
     4x x  x x  + 4x x  x  + 3x x x x  + 4x x x  + 3x x x x
       0 2  1 2     0 2  1     1 0 2 1     1 0 2      1 2 0 1
   +
            3          2          2          3          2  2
     4x x x x   + 3x x x x x  + 3x x x  x x  + 3x x x  x  + 4x x  x x
       1 2 0 2      1 2 1 0 1     1 2 1  0 1     1 2 1  0     1 2  0 2
   +
         3          4          4          3          2
     4x x  x x  + 4x x  x  + 12x x x  + 4x x x x  + 4x x x x x
       1 2 0 2      1 2  0     2 0 1     2 0 1 2     2 0 2 1 2

   +
```

```
          2             3             3           3         2   2
  4x x x  x x  + 4x x x  x  + 12x x x x  + 4x x x x  + 12x x  x x
    2 0 2  1 2    2 0 2  1     2 1 0 1     2 1 0 2      2 1  0 1
+
         3           4             2         2            3
  12x x  x x  + 12x x  x  + 4x x x x x  + 4x x x  x x + 4x x x  x
    2 1  0 1     2 1  0    2 1 2 0 2     2 1 2  0 2    2 1 2  0
+
     2     2     2            2   2       2     2
  4x  x x x  + 4x  x x x x + 4x  x x  x + 4x  x x x
    2  0 1 2     2  0 2 1 2    2  0 2  1    2  1 0 2
+
     2            2   2       3           3           3
  4x  x x x x + 4x  x x  x + 4x  x x x + 4x  x x x + 4x  x x x
    2  1 2 0 2    2  1 2  0    2  0 1 2    2  0 2 1    2  1 0 2
+
     3          4         4          2 5          4
  4x  x x x + 4x  x x + 4x  x x - 10x  x  - 6x x x x
    2  1 2 0    2  0 1    2  1 0     0  1     0 1 0 1
+
          2   3       3   2       4           5         2 4
  - 5x x  x x  - 5x x  x x  - 5x x  x x - 5x x  x  - 2x x  x
      0 1  0 1     0 1  0 1     0 1  0 1    0 1  0    1 0  1
+
              3        2   2        3           4
  - x x x x x  - x x x  x x  - x x x  x x - x x x  x
     1 0 1 0 1    1 0 1  0 1    1 0 1  0 1   1 0 1  0
       Type: XDistribuedPolynomial(IndexedVariable x,RationalNumber)
                                  Time: .1 (EV) + .6 (OT) = .7 sec


;sd:sdyn := realisation(g);      -- algo de realisation
   Mots de Lyndon
   Calcul de la base-standard des residuels de g
   Rang de Lie: 7
   Calcul des coordonnees locales
   Calcul de la fonction d'observation
   Calcul des champs de vecteurs
                                                        Type: Void
                                  Time: .1 (IN) + 54.3 (EV) = 54.4 sec


;dimension sd            -- dimension de l'espace d'etat

   (12)  7
                                                Type: PositiveInteger
                                  Time: .067 (OT) = .067 sec
```

```
;Observation sd          -- fonction d'observation

        1    5  2     1      4   1      3   1      2   1
  (13)  --- q  q   + (--- q q   + -- q q   + -- q q   + - q q )q
        600  5  6     120  3 5    30  2 5    10  1 5    5  0 5  6
 Type: SparseMultivariatePolynomial(RationalNumber,IndexedVariable q)
                                          Time: .1 (EV) + .1 (OT) = .2 sec

;VectField(sd,x0)        -- champ de vecteurs obtenu pour chaque entree

            d        d
  (14)  q  ---- - 5 ----
           5 d q      d q
              3        6
Type: VectorField(IndexedVariable q,SMP(RationalNumber,IndexedVariable q))
                                                  Time: .1 (OT) = .1 sec

;VectField(sd,x1)

            d          d          d          d      d
  (15)  q  ---- + q  ---- + q  ---- + q  ---- - ----
           1 d q      2 d q      3 d q      6 d q    d q
              0          1          2          3      5
Type: VectorField(IndexedVariable q,SMP(RationalNumber,IndexedVariable q))
                                                Time: .067 (OT) = .067 sec

;VectField(sd,x2)

        1    3  d          d          d          d
  (16)  -- q   ---- - 3q  ---- + 12 ---- + 4 ----
        96   4  d q        5 d q      d q        d q
                  0          2          3          4
Type: VectorField(IndexedVariable q,SMP(RationalNumber,IndexedVariable q))
                                                  Time: .1 (EV) = .1 sec
```

# F

# Le groupe de Lie

## F.1 Le fichier source

Voici le contenu d'un fichier permettant d'effectuer sous interprèteur SCRATCHPAD quelques calculs dans le groupe de Lie d'une algèbre nilpotente à l'ordre 4. On observe que les coefficients $\{c_0, c_1\}$ sont des variables commutatives tandis que les lettres $\{x_0, x_1\}$ ne commutent pas. Ce genre de calcul est délicat à effectuer dans les systèmes de calcul formel non typés.

```
)clear all

COEF := P RN                        -- definition des domaines
lettre := OrderedVarlist1 [x0,x1]
GL   := LieExponentials([x0,x1], COEF, 4)
LP   := LiePolynomial(lettre, COEF)
PBW  := XPBWPolynomial(lettre, COEF)

x0 :lettre := variable('x0)         -- forcage de type
x1 :lettre := variable('x1)

p0 : LP :=  x0                      -- debut des calculs
p0 := c0::COEF * p0
g0 : GL := exp(p0)

p1 : LP := x1
p1 := c1::COEF * p1
g1 : GL := exp(p1)

g : GL := g0 * g1
g :: PBW                            -- representation interne

log(g)$GL
mirror(g)$GL
h: GL := inv(g)$GL
h * g
g * h
```

## F.2 Trace d'exécution

On constate sur cet exemple que les éléments du groupe de Lie sont affichés sous forme de produits décroissant d'exponentielles mais sont repésentés de façon interne par des polynômes exprimés dans la base $PBWL$. Ne pas trop tenir compte des temps de calcul qui fluctuent assez sensiblement d'une exécution à l'autre.

```
;COEF := P RN

   (1)  Polynomial RationalNumber
                                                          Type: Domain
                                                           Time: 0 sec

;lettre := OrderedVarlist1 [x0,x1]

   (2)  OrderedVarlist1 [x0,x1]
                                                          Type: Domain
                                        Time: .3 (IN) + .2 (OT) = .5 sec

;GL    := LieExponentials([x0,x1], COEF, 4)

   (3)  LieExponentials([x0,x1],Polynomial RationalNumber,4)
                                                          Type: Domain
                                    Time: .067 (IN) + .067 (OT) = .133 sec

;LP    := LiePolynomial(lettre, COEF)

   (4)
   LiePolynomial(OrderedVarlist1 [x0,x1],Polynomial RationalNumber)
                                                          Type: Domain
                                               Time: .067 (IN) = .067 sec

;PBW   := XPBWPolynomial(lettre, COEF)

   (5)
   XPBWPolynomial(OrderedVarlist1 [x0,x1],Polynomial RationalNumber)
                                                          Type: Domain
                                                   Time: .1 (OT) = .1 sec

;x0 :lettre := variable('x0)

   (6)  x0
                                             Type: OrderedVarlist1 [x0,x1]
                                          Time: .1 (IN) + .1 (OT) = .2 sec

;x1 :lettre := variable('x1)

   (7)  x1
                                             Type: OrderedVarlist1 [x0,x1]
```

```
;p0 : LP :=  x0

   (8)  {x0}
Type: LiePolynomial(OrderedVarlist1 [x0,x1],Polynomial RationalNumber)
                                        Time: .1 (IN) + .1 (OT) = .2 sec
;p0 := c0::COEF * p0

   (9)  c0{x0}
Type: LiePolynomial(OrderedVarlist1 [x0,x1],Polynomial RationalNumber)
                                    Time: .3 (IN) + .067 (OT) = .367 sec
;g0 : GL := exp(p0)

           c0{x0}
   (10)  e
            Type: LieExponentials([x0,x1],Polynomial RationalNumber,4)
                                    Time: .033 (IN) + .1 (EV) = .133 sec
;p1 : LP := x1

   (11)  {x1}
Type: LiePolynomial(OrderedVarlist1 [x0,x1],Polynomial RationalNumber)
                                        Time: .1 (IN) + .1 (OT) = .2 sec

;p1 := c1::COEF * p1

   (12)  c1{x1}
Type: LiePolynomial(OrderedVarlist1 [x0,x1],Polynomial RationalNumber)
                                    Time: .2 (IN) + .067 (OT) = .267 sec

;g1 : GL := exp(p1)

           c1{x1}
   (13)  e
            Type: LieExponentials([x0,x1],Polynomial RationalNumber,4)
                                    Time: .067 (IN) + .1 (OT) = .167 sec

;g : GL := g0 * g1

   (14)
                 1      3      3  1      2      2
                 - c0 c1 {x0 x1 } - c0 c1 {x0 x1 }
      c1{x1} 6                   2                  c0 c1{x0 x1}
     e      e                  e                  e
  *
      1   2  2   2  2  1   2     2    1   3     3
      - c0 c1 {x0 x1 } - c0 c1{x0 x1} - c0 c1{x0 x1}
      4                2             6              c0{x0}
     e                e             e              e
            Type: LieExponentials([x0,x1],Polynomial RationalNumber,4)
                                Time: .3 (IN) + .3 (EV) + .1 (OT) = .7 sec
```

```
;g :: PBW                        -- representation interne

   (15)
                                1   2
     1 + c0{x0} + c1{x1} + 1/2 c0 {x0}{x0} + c0 c1{x0 x1} + c0 c1{x1}{x0}
   +
     1   2           1   3               1   2     2
     - c1 {x1}{x1} + - c0 {x0}{x0}{x0} + - c0 c1{x0 x1}
     2               6                   2
   +
       2                 1      2      2    1   2
     c0 c1{x0 x1}{x0} + - c0 c1 {x0 x1 } + - c0 c1{x1}{x0}{x0}
                       2                   2
   +
        .............
   +
     1      3                     1    4
     - c0 c1 {x1}{x1}{x1}{x0} + -- c1 {x1}{x1}{x1}{x1}
     6                           24
Type: XPBWPolynomial(OrderedVarlist1 [x0,x1],Polynomial RationalNumber)
                                              Time: .067 (OT) = .067 sec
;log(g)$GL

   (16)
                          1                  1    2     2
     c0{x0} + c1{x1} + - c0 c1{x0 x1} + -- c0 c1{x0 x1}
                          2                  12
   +
     1       2      2    1    2  2   2  2
     -- c0 c1 {x0 x1 } + -- c0 c1 {x0 x1 }
     12                   24
Type: LiePolynomial(OrderedVarlist1 [x0,x1],Polynomial RationalNumber)
                       Time: .1 (IN) + 3.833 (EV) + .067 (OT) = 4.0 sec
;mirror(g)$GL

           c1{x1} c0{x0}
   (17)  e      e
           Type: LieExponentials([x0,x1],Polynomial RationalNumber,4)
                                    Time: .067 (IN) + .6 (EV) = .667 sec
;h: GL := inv(g)$GL

         - c1{x1} - c0{x0}
   (18)  e        e
           Type: LieExponentials([x0,x1],Polynomial RationalNumber,4)
                           Time: .1 (IN) + .6 (EV) + .1 (OT) = .8 sec
;h * g
```

```
   (19)  1
                Type: LieExponentials([x0,x1],Polynomial RationalNumber,4)
                           Time: .2 (IN) + 10.3 (EV) + .1 (OT) = 10.6 sec

;g * h

   (20)  1
                Type: LieExponentials([x0,x1],Polynomial RationalNumber,4)
                                  Time: .167 (IN) + 1.433 (EV) = 1.6 sec
```

## F.3 Produit de deux séries de Chen

Voici les formules calculées en SCRATCHPAD permettant de calculer la série de Chen correspondant à la concaténation de deux entrées quelconques (voir 7.4.1.5 page 107), les calculs étant valides pour une algèbre de Lie nilpotente à l'ordre 5.

```
====> {x1}
b   + a
 13    13
:::::::::::::::::
             4
====> {x0 x1 }
1        4   1       3   1        2
-- a b    + - a b    + - a  b    + a  b   + b   + a
24  0 13    6  8 13    2  10 13     11 13    12    12
:::::::::::::::::
             3
====> {x0 x1 }
1       3   1      2
- a b    + - a b    + a  b   + b   + a
6  0 13    2  8 13     10 13    11    11
:::::::::::::::::
             2
====> {x0 x1 }
1       2
- a b    + a b   + b   + a
2  0 13     8 13    10    10
:::::::::::::::::
                  2
====> {x0 x1 x0 x1 }
  1   2   3    1         1        2            1   2
  - a  b    + (- a a  + - a )b    + (a b   + - a   + a )b   + a b
  4  0  13     2  0 8   2  4  13      0 10   2  8     6  13    8 10
+
  b  + a
   9    9
:::::::::::::::::
====> {x0 x1}
```

```
a b   + b  + a
 0 13    8    8
::::::::::::::::
         2  3
====> {x0 x1 }
1    2   3   1      2
-- a  b    + - a b    + a b   + a b   + b  + a
12  0  13    2  4 13     6 13    0 11    7    7
::::::::::::::::
         2  2
====> {x0 x1 }
1   2   2
- a  b    + a b   + a b   + b  + a
4  0  13     4 13    0 10    6    6
::::::::::::::::
         2
====> {x0 x1 x0 x1}
1   3   2    1   2                        1     2
- a  b    + (- a  b  + a a  - a )b   + - a b   + a b  + b  + a
6  0  13     2  0  8    0 4    2  13   2  0 8     4 8    5    5
::::::::::::::::
         2
====> {x0 x1}
1   2
- a  b   + a b  + b  + a
2  0  13    0 8    4    4
::::::::::::::::
         3  2
====> {x0 x1 }
1    3   2           1   2
-- a  b    + a b   + - a  b   + a b  + b  + a
12  0  13     2 13   2  0  10    0 6    3    3
::::::::::::::::
         3
====> {x0 x1}
1   3      1   2
- a  b   + - a  b  + a b  + b  + a
6  0  13   2  0  8    0 4    2    2
::::::::::::::::
         4
====> {x0 x1}
1    4      1   3     1   2
-- a  b   + - a  b  + - a  b  + a b  + b  + a
24  0 13   6  0  8   2  0  4    0 2    1    1
::::::::::::::::
====> {x0}
b  + a
 0    0
```

## Abstract

We study some fundamental properties of nonlinear dynamic systems with a view to have effective algorithms and to implemente them in a hight level computer algebra system.

The minimal analytic realization and the motin planning problems are approached using noncommutative formal power series.

We prove by a constructive method that *any polynomial generating power series has a polynomial minimal realization.*

In this thesis, we give some basic algorithms which permit to implement the noncommutative polynomials, the Lie polynomials, the Lyndon words and Lyndon polynomials, the free nilpotent Lie groups, and other technical tools implementation in SCRATCHPAD–AXIOM.

Four representations for a noncommutative polynomial are given:

- distributed,
- recursive,
- in a Poincaré–Birkhoff–Witt basis,
- in a transcendance basis of the shuffle algebra.

A complete package is given computing the minimal realization of polynomial generating power series.

Some basic tools are presented, involved for the treatment of the motion planning problem actually developped in the S.N.C.F. research group.

The algorithms are implemented in SCRATCHPAD–AXIOM using the technics of software engineering: genericity, abstract types and inheritance.



**Key words:** Nonlinear control, Algebraic computing, Analytic minimal realization, Lie algebra, Noncommutative polynomials, Shuffle algebra.

## Résumé

On étudie quelques propriétés fondamentales des systèmes dynamiques non linéaires dans le but d'avoir des algorithmes effectifs et de les implanter dans un système de calcul formel de haut niveau.

Le problème de la réalisation analytique minimale et celui du "motion planning" sont abordés par le biais des séries formelles en variables non commutatives.

On démontre, de manière constructive, qu'*une série génératrice polynomiale admet une réalisation minimale polynomiale.*

On présente les algorithmes de base permettant l'implantation des polynômes en variables non commutatives, les polynômes de Lie, les mots et les polynômes de Lyndon, le groupe de Lie nilpotent libre et d'autres outils techniques dans un système de calcul formel.

Quatre représentations des polynômes non commutatifs sont données :

- distribuée,
- récursive,
- dans une base de Poincaré–Birkhoff–Witt,
- dans une base de transcendance de l'algèbre de mélange.

Un paquetage complet pour le calcul de la réalisation analytique minimale des séries génératrices polynomiale est donné ainsi que des outils de base pour le traitement du "motion planning", actuellement développé dans l'équipe S.N.C.F.

Les algorithmes sont implantés dans le système de calcul formel SCRATCHPAD–AXIOM en utilisant les techniques du génie logiciel : généricité, types abstraits, héritage.


**Mots clé :** Algèbre de Lie, Algèbre de mélange, Automatique non linéaire, Calcul formel, Polynôme non commutatifs, Realisation analytique minimale.