Polynômes non commutatifs : représentation et traitement par les systèmes de Calcul Formel*

N.E. Oussous et M. Petitot

Laboratoire d'Informatique Fondamentale de Lille
U.A. 369 du C.N.R.S., Université de Lille I,
59655 Villeneuve d'Ascq Cedex. FRANCE.

Résumé : Dans ce papier, après quelques rappels de base sur les mots, polynômes, séries formelles et opérations définies dessus, on montre que la représentation interne des polynômes conditionne l'efficacité des algorithmes. La plupart de ces algorithmes sont implantés en *Macsyma* et en *Scratchpad.* Ce papier est un exemple d'utilisation des concepts modernes en génie logiciel : types abstraits et programmation orientée objets.

Abstract : In this paper, we present in first some basic reminds on words, polynomials, formal power series and some related operations. After that, under our experience with the two computer algebra systems *Macsyma* and *Scratchpad,* we show that the internal representation of polynomials affects the algorithm performance. This paper is an example of use of the modern concepts in software engineering: Abstract Data Types and Object Oriented Programming.

1 Introduction

Ce travail s'inscrit dans le cadre d'un projet de recherche sur les séries formelles non commutatives et les systèmes dynamiques non linéaires. Nous y traitons plus particulièrement les polynômes non commutatifs et nous espérons des extensions aux séries rationnelles.

Dans ce papier, nous abordons le problème de la représentation des polynômes non commutatifs dans les systèmes de calcul formel. Ainsi, en *Macsyma,* nous utilisons deux opérateurs prédéfinis :

- $*$: multiplication d'un mot par un coefficient (commutative),
- $\cdot$: multiplication de deux mots ou de deux polynômes (non commutative).

* *Ce travail est pratiellement supporté par IBM France et par le PRC Mathématiques et Informatique. Il est effectué dans le cadre d'une étude jointe avec IBM Yorktown, USA.*

Toute expression construite à partir de ces deux opérateurs peut être réduite par le "simplificateur" prédéfini de *Macsyma.*

Ainsi, on travaille directement sur la représentation interne utilisée par *Macsyma* pour toutes les expressions rationnelles en réutilisant, telles quelles, les opérations prédéfinies comme la multiplication et l'addition. L'implantation des autres fonctions (résiduel, shuffle) est programmée récursivement en testant l'opérateur situé à la racine des polynômes. On utilise les fonctions offertes par *Macsyma* à savoir `part` et `inpart` pour accéder aux différentes parties d'une expression. Il se trouve alors que l'exploration de l'arbre d'une expression ainsi que les tests sur les noeuds sont une opération assez coûteuse.

En *Scratchpad,* on a adopté une représentation, dite récursive, qui consiste à représenter un polynôme par un couple de deux termes : le terme constant du polynôme et la liste de ses résiduels par les différentes lettres de l'alphabet :

$$P \longrightarrow (\langle P|\varepsilon\rangle, ((x_1, P \triangleright x_1), \ldots)).$$

Les résiduels étant des polynômes, ils sont donc représentés de la même façon. Il faut écrire complètement toutes les opérations sur les polynômes, y compris les opérations de base comme l'addition ou la multiplication.

Une deuxième représentation, dite distribuée, est aussi possible. Elle consiste à représenter un polynôme par la liste de ses monômes. Les deux représentations sont implantées et pour les calculs les plus coûteux, à savoir le produit de mélange, il se trouve que la représentation récursive est de loin la meilleure.

On aurait pu imaginer une représentation récursive en *Macsyma,* mais celle-ci aurait été plus lourde d'utilisation car *Macsyma* ne permet pas la surcharge des opérateurs.

2 Rappels et définitions

2.1 Alphabet et mots

Soit $X = \{x_0, x_1, ..., x_m\}$, un ensemble de *lettres* appelé *alphabet.* Un *mot* de longueur n est une suite $u = x_{i_1} x_{i_2} \ldots x_{i_n}$ de n lettres de X. Le *monoïde libre* X^* est l'ensemble des mots sur l'alphabet X muni de la *concaténation.* On notera $u.v$ ou uv la concaténation de deux mots u et v. Le mot *vide*, noté ε, est le mot qui ne contient aucune lettre. On notera $|w|$ la longueur du mot w. Ainsi la longueur du mot vide est nulle.

2.2 Séries et polynômes non commutatifs

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

$$S = \sum_{w \in X^*} \langle S|w\rangle w.$$

L'ensemble des séries formelles sera noté $\mathbb{R}\langle\langle X\rangle\rangle$.

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

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

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

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 $\mathbb{R}\langle X\rangle$, c'est une sous-algèbre de $\mathbb{R}\langle\langle X\rangle\rangle$.

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

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

2.3 Opérations

2.3.1 Produit de Cauchy

Le *produit de Cauchy* pour les séries formelles est une extension du produit de concaténation défini sur les mots. Soient $S, T \in \mathbb{R}\langle\langle X\rangle\rangle$, deux séries formelles non commutatives. Le produit de Cauchy de S par T, noté $S \cdot T$ ou tout simplement ST est défini par :

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

Ce qui peut aussi s'écrire : $\forall S, T \in \mathbb{R}\langle\langle X\rangle\rangle$, $\forall w \in X^*$,

$$\langle S \cdot T|w\rangle = \sum_{u \cdot v = w} \langle S|u\rangle\langle T|v\rangle.$$

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.*

2.3.2 Produit de mélange

Le *produit de mélange*[1] (M.Fliess [3]; G.Jacob & N.Oussous [4]) est défini récursivement, pour les mots, par :

$$\begin{cases} \forall w \in X^*, & w \sqcup \varepsilon = \varepsilon \sqcup w = w, \quad (\varepsilon \text{ étant le mot vide}). \\ \forall u, v \in X^*,\ \forall x, y \in X, & (xu) \sqcup (yv) = x(u \sqcup (yv)) + y((xu) \sqcup v). \end{cases} \tag{1}$$

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.

[1] En anglais : Shuffle product.

Ce produit est commutatif et associatif et peut s'étendre facilement aux polynômes et aux séries en posant, pour $S, T \in \mathbb{R}\langle\langle X \rangle\rangle$ des séries :

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

Remarque 2.1 *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.*

On peut étendre, par linéarité, l'algorithme de calcul du produit de mélange des mots à celui des polynômes. Mais, cet algorithme n'est pas efficace. On verra par la suite qu'avec une bonne maîtrise de la représentation interne, on peut obtenir un algorithme performant.

2.3.3 Calcul des résiduels

Définition 2.1 *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 :*

$$u \triangleright x = \begin{cases} v & si \quad u = xv, \\ 0 & sinon. \end{cases} \qquad \left(resp.\ x \triangleleft u = \begin{cases} v & si \quad u = vx, \\ 0 & sinon. \end{cases} \right) \tag{2}$$

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

$$w \triangleright u = \begin{cases} v & \text{si} \quad w = uv, \\ 0 & \text{sinon.} \end{cases} \qquad \left(\text{resp. } u \triangleleft w = \begin{cases} v & \text{si} \quad w = vu, \\ 0 & \text{sinon.} \end{cases} \right) \tag{3}$$

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

$$S \triangleright u = \sum_{u \in X^*} \langle S|w\rangle w \triangleright u \qquad \left(\text{resp. } u \triangleleft S = \sum_{u \in X^*} \langle S|w\rangle u \triangleleft w \right) \tag{4}$$

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

$$S \triangleright P = \sum_{u \in X^*} \langle P|u\rangle S \triangleright u \qquad \left(\text{resp. } P \triangleleft S = \sum_{u \in X^*} \langle P|u\rangle u \triangleleft S \right) \tag{5}$$

Le produit de mélange défini dans le paragraphe précédent, peut être redéfini en posant, pour des mots u et v et une lettre z :

$$(u \sqcup v) \triangleright z = (u \triangleright z) \sqcup v + u \sqcup (v \triangleright z), \tag{6}$$

$$\langle u \sqcup v|\varepsilon\rangle = \begin{cases} 0 & \text{si} \quad uv \neq \varepsilon, \\ 1 & \text{si} \quad uv = \varepsilon. \end{cases}$$

Ainsi, cette action est une dérivation pour le produit de mélange. On peut donc la noter : $\partial_z \equiv \triangleright z$.

Lemme 2.1 *Si $z \in X$, alors ∂_z est une dérivation dans $\mathbb{R}\langle\langle X\rangle\rangle$ pour le produit de mélange.*

Preuve : Soient $S, T \in \mathbb{R}\langle\langle X\rangle\rangle$.

$$\begin{aligned}
\partial_z(S \sqcup T) &= \sum_{u,v\in X^*} \langle S|u\rangle\langle T|v\rangle \partial_z(u \sqcup v) \\
&= \sum_{u,v\in X^*} \langle S|u\rangle\langle T|v\rangle \left[(\partial_z u)\sqcup v + u \sqcup (\partial_z v)\right] \\
&= \sum_{u,v\in X^*} \langle S|u\rangle\langle T|v\rangle (\partial_z u)\sqcup v + \sum_{u,v\in X^*} \langle S|u\rangle\langle T|v\rangle u \sqcup (\partial_z v) \\
&= \left(\sum_{u\in X^*} \langle S|u\rangle (\partial_z u)\right) \sqcup \left(\sum_{v\in X^*} \langle T|v\rangle v\right) \\
&\qquad + \left(\sum_{u\in X^*} \langle S|u\rangle u\right) \sqcup \left(\sum_{v\in X^*} \langle T|v\rangle (\partial_z v)\right) \\
&= (\partial_z S)\sqcup T + S \sqcup (\partial_z T).
\end{aligned}$$

Ce lemme est valable aussi pour les polynômes. De plus, on a, pour des séries S et T et une lettre z, la propriété suivante :

$$\partial_z(S \cdot T) = (\partial_z S)\cdot T + \langle S|\varepsilon\rangle \partial_z T. \tag{7}$$

2.3.4 Lemme de reconstruction

Ayant les restes de S à droite par les lettres $z \in X$ et le terme constant $\langle S|\varepsilon\rangle$ de S, on peut reconstruire S. On a le lemme suivant dit *lemme de reconstruction.*

Lemme 2.2 [2] *Soit $S \in \mathbb{R}\langle\langle X\rangle\rangle$ une série formelle et soit $\langle S|\varepsilon\rangle$ son terme constant. Alors*

$$\begin{aligned}
S &= \langle S|\varepsilon\rangle + \sum_{z\in X} z(\partial_z S) \\
&= \langle S|\varepsilon\rangle + \sum_{z\in X} z(S \triangleright z).
\end{aligned}$$

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

3 Les systémes de calcul formel

Les systèmes de calcul formel (SCF) sont de grands logiciels qui calculent sur les nombres, les symboles et les formules. Ils sont devenus d'importants outils de recherche.

[2] Il existe un lemme de reconstruction plus général dû à Brzozowski et *al.*

Dans nos projets de recherche, nous faisons beaucoup de calculs symboliques et algébriques et nous sommes amenés à utiliser les SCF. Dans ce qui suit, nous rappelons notre expérience avec *Macsyma* et nous détaillons notre travail avec *Scratchpad.*

Nous avons choisi de parler spécialement de ces deux systèmes d'abord parce que ce sont les deux systèmes que nous avons le plus utilisés, mais aussi parce que *Macsyma* appartient à la grande classe des sytèmes *non typés* tout comme *Reduce, Maple, Mathematica* et d'autres et que *Scratchpad* est le seul système *typé.*

3.1 Le système Macsyma

Le système *Macsyma* est dans la classe des systèmes non typés. Dans cette classe, on trouve les systèmes classiques tels que *Reduce,* dont une introduction est donnée dans [2], *Mumath* et moins classiques tels que *Maple,* dont une introduction est donnée dans [1] et *Mathematica.*

Macsyma est le géant des systèmes de cette classe. Il a été créé dans les années soixante-dix. Il tourne sur les grosses machines. Il possède une très grande bibliothèque de fonctions. Il a une syntaxe très simple (proche de celle du Pascal). On trouvera une introduction à ce système dans Pavelle and Wang [10].

Nous disposons de *Macsyma* sur *Sun.* Nous avons développé des outils de manipulations et de traitement des polynômes non commutatifs. Nous avons utilisé l'opérateur "." de *Macsyma* pour le produit de concaténation. Ainsi, le mot $w = abcab$ sera implanté par $a \cdot b \cdot c \cdot a \cdot b$ et sa représentation interne sera alors l'arbre donné par la figure 1–(1). Un polynôme sera considéré comme une combinaison linéaire de mots, $P = \sum_{i=1}^{n} a_i * w_i$, et sa représentation interne sera l'arbre donné par la figure 1–(2). On utilise alors les fonctions "`part`" et "`inpart`" de *Macsyma* pour explorer ces arbres. On a implanté les différentes opérations (G.Jacob et N.E.Oussous [5]) : produit de mélange, calcul des résiduels, calcul du degré...etc. Ensuite, on a développé un paquetage de programmes qui permettent de calculer la réalisation des systèmes dynamiques non linéaires de série génératrice finie (G.Jacob et N.E.Oussous [6]; N.E.Oussous [7, 8]). Malheureusement, en traitant les polynômes non commutatifs, on s'est apperçu que pour le produit de mélange, les temps de calcul et la place mémoire occupée sont prohibitifs. Ceci est essentiellement dû à la représentation interne et à la façon dont on l'explore. On aurait pu utiliser une représentation récursive (voir section 4.3) mais, *Macsyma* n'est pas typé et ne permet pas la surcharge des opérateurs et donc une telle représentation serait très lourde d'utilisation. A la lumière de notre experience, une perte de temps est previsible en mode interprété. Pour toutes ces raisons nous nous sommes tournés vers *Scratchpad.*

3.2 Le système *Scratchpad*

Nous allons tenter de résumer en quelques lignes les principaux concepts qui sont à l'oeuvre dans le système *Scratchpad* :

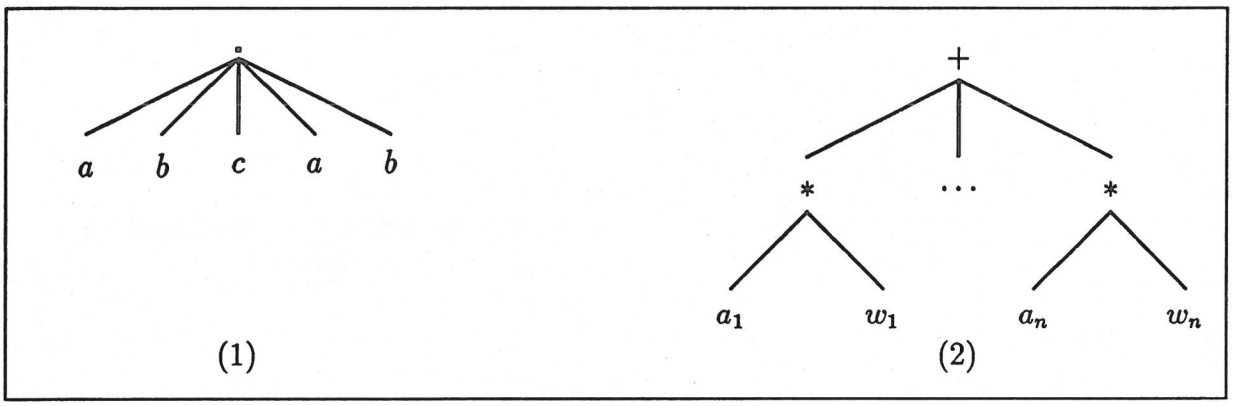

Figure 1: Représentations internes en *Macsyma.*

1. C'est un langage de programmation à typage *implicite;* le type d'une variable peut ne pas être déclaré; il est alors inféré lors de sa première affectation.

2. Ce langage est supporté à la fois par un interpréteur et un compilateur se servant de Common Lisp comme langage intermédiaire. Les instructions exécutées en mode interprété et compilé sont les mêmes; les différences de fonctionnnement sont assez minimes.

3. Les *unités de compilation* sont de trois sortes :

   (a) Les paquetages,

   (b) Les constructeurs de domaines,

   (c) Les constructeurs de catégories,

4. Un type (domaine) est défini par :

   (a) sa *partie publique* qui est une catégorie,

   (b) sa *partie privée* comportant une description de la représentation interne des objets du domaine ainsi que l'implantation des fonctions assurant la gestion de ces objets.

5. Une catégorie est la donnée :

   (a) d'une liste de fonctions et d'opérations. Les types des paramètres formels en entrée et des résultats retournés par ces fonctions sont également précisés (signatures des fonctions).

   (b) d'une liste d'attributs précisant la sémantique de ces fonctions et opérations.

Exemple 3.1

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
```

---

6. Les constructeurs de domaines fournissent généralement des constructions génériques (ie. paramétrées par d'autres types). Ainsi le constructeur :

   `SparseMultivariatePolynomial(R : Ring, VarSet : OrderedSet)`

   gère les polynômes formels à plusieurs variables en représentation récursive creuse. Ces polynômes ont leurs coefficients dans un anneau quelconque `R` et les variables sont prises dans un ensemble ordonné quelconque `varSet`.
   Nous observons l'utilisation de la notion de catégorie (`Ring` et `OrderedSet` sont des catégories) pour assurer un contrôle des paramètres génériques requis.

7. *Scratchpad* est un *environnement de programmation* qui permet de visualiser la partie publique de l'ensemble des unités de compilation. Ceci est essentiel pour la réutilisation des composants logiciels existants.

4 L'implantation des polynômes non commutatifs

4.1 Harmoniser l'interface

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 qui se trouve être critique) mais **d'identifier clairement les objets à implanter et de fixer leur interface** (ie. donner un nom au fonctions qu'ils exportent).

Nous avons étudié les principales opérations permettant de gérer les polynômes formels non commutatifs à savoir:

1. opérations habituelles sur les algèbres,

2. résiduels à gauche et à droite,

3. produit de mélange (shuffle),

4. problèmes d'algèbre linéaire: dépendance linéaire, bases, coordonnées.

Il se trouve que l'étude des systèmes dynamiques non linéaires requiert l'utilisation d'autres outils tels que séries formelles rationnelles et R-automates, et que ces objets exportent les mêmes opérations de base.

Il est utile d'harmoniser l'interface en définissant une catégorie commune.

XFreeAlgebra

```
XFreeAlgebra(vl:OrderedSet,R:Ring):Category == Catdef where
   Catdef == Join(Ring,Algebra(R)) with
     -- exports
      "*": (vl,$) -> $
      mindeg: $ -> OFMON1(vl)          ++ Attention pour le polynome nul !!!
      coef  : ($,OFMON1(vl)) -> R      ++ coefficient d'un mot
      lquo  : ($,vl) -> $              ++ residuel par une lettre
      lquo  : ($,OFMON1(vl)) -> $      ++ residuel par un mot
      rquo  : ($,vl) -> $
      rquo  : ($,OFMON1(vl)) -> $
      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 :($,$) -> $                   ++ shuffle product
      map   : (R -> R, $) -> $         ++ transf. des coef par une fonction
      varList: $ -> List vl            ++ liste des variables
     -- attributs
      if R has noZeroDivs then noZeroDivs
      if R has Field then "/" : ($,R) -> $
```

Il nous faut maintenant réfléchir sur la représentation interne des polynômes non commutatifs; nous verrons que ce choix conditionne de façon importante l'efficacité de l'implantation au niveau du temps de calcul et de la place mémoire occupée.

4.2 La représentation distribuée

4.2.1 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:

OrderedFreeMonoid1

```
)abbrev domain OFMON1 OrderedFreeMonoid1
OrderedFreeMonoid1(S: OrderedSet): OFMcategory == OFMdefinition where
    NNI ==> NonNegativeInteger
    OFMcategory == OrderedMonoid with
        coerce: S -> $
        "*":    (S, $) -> $
        "**":   (S, NNI) -> $
        first: $ -> S
        rest:  $ -> $
        ...
    OFMdefinition == FreeMonoid(S) add
      -- representation
        Term := Record(gen: S, exp:Integer )
        Rep:= List Term
      -- definitions
        first x ==
          null x => error "empty word !!!"
          x.0.gen
        rest x ==
          null x => error "empty word !!!"
          x.0.exp = 1 => rest$Rep x
          cons([x.0.gen,x.0.exp-1],rest$Rep x)
        a < b ==                -- ordre lexicographique par longueur
            la:NNI :=length a; lb:NNI:=length b
            la < lb =>  true
            la > lb =>  false
            lexico(a,b)
        ...
```

On constate sur cet exemple que l'on a facilement réutilisé le constructeur de domaines FreeMonoid(S) en lui ajoutant quelques fonctionnalités qui nous manquaient, en particulier l'*ordre lexicographique par longueur.*

4.2.2 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 correspondant au concept mathématique d'*algèbre de monoïde.*

XPolynomialRing

```
)abbrev domain   XPR XPolynomialRing
XPolynomialRing(R:Ring,E:OrderedMonoid): T == C where
  T == Join(Ring,Algebra(R)) with
    --operations
      "#": $ -> NonNegativeInteger
      coerce: E -> $
      maxdeg: $ -> E
      mindeg: $ -> E
      ...
      if R has Field then "/" : ($,R) -> $
    --assertions
      if R has noZeroDivs then noZeroDivs
      if R has unitsKnown then unitsKnown
      if R has canonicalUnitNormal then canonicalUnitNormal
  C == FreeModule(R,E) add
    --representations
       Term:=  Record(k:E,c:R)
       Rep:=  List Term
    --define
       1  == [[1$E,1$R]]
       #x == #$Rep x
       maxdeg p == if null p then 1$E else p.first.k
       mindeg p == if null p then 1$E else (last p).k
       ...
```

Cette unité de compilation est la copie d'une autre existant pour l'algèbre commutative; malheureusement, elle doit être entièrement réécrite à cause d'une faiblesse de conception en *Scratchpad.* Le concept de catégorie y est sensiblement éloigné du concept mathématique : en *Scratchpad*, un monoïde abélien n'est pas un monoïde car dans un cas, la loi de composition est notée "+" et dans l'autre "$*$".

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.

4.2.3 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:

XDistribuedPolynomial

```
)abbrev domain XDPOLY XDistribuedPolynomial

OFMON1 ==> OrderedFreeMonoid1
XDistribuedPolynomial(vl:OrderedSet,R:Ring): XDPcat == XDPdef where
  XDPcat == XFreeAlgebra(vl,R) with
    maxdeg: $ -> OFMON1(vl)
    degree: $ -> NonNegativeInteger
    if vl has Finite then
       Vectorise   : $ -> Vector R
       UnVectorise : Vector R -> $
  XDPdef == XPolynomialRing(R,OFMON1(vl)) add
    -- declarations
       word := OFMON1(vl)
       x,y,v:vl
       ...
    -- Representation
       Term:=  Record(k:word,c:R)
       Rep:=  List Term
    -- local functions
       shw: (OFMON1 vl, OFMON1 vl) -> $    -- shuffle de 2 mots
    -- definitions
       v * p == [[v * t.k , t.c]$Term for t in p]
       ...
```

4.3 La représentation récursive

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

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

$\sum_{x \in X} x(p \triangleright x)$ 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.

4.3.1 Formellement:

Un polynôme de $R\langle X\rangle$ est soit:

- un élément de l'anneau R
- un *vrai polynôme*

Un *vrai polynôme* comporte:

- un terme constant (qui peut être nul),
- une partie propre qui appartient à un *module libre* construit sur l'alphabet X, les coefficients étant des polynômes.

Tout ceci se traduit très naturellement en *Scratchpad* [3]:

XRecursivePolynomial

```
)abbrev domain XRPOLY      XRecursivePolynomial

XDPOLY ==> XDistribuedPolynomial
XRecursivePolynomial(VarSet:OrderedSet,R:Ring):  Xcat == Xdef
  where
  Xcat == XFreeAlgebra(VarSet,R) with
       extend: $ -> XDPOLY(VarSet,R)
       maxdeg: $ -> OrderedFreeMonoid1 VarSet
       degree: $ -> NonNegativeInteger
  Xdef == add
    -- representation
       RegPoly := FreeModule1($,VarSet)
       VPoly   := Record(c0:R,reg:RegPoly)
       Rep     := Union(R,VPoly)
    --define
       0 == 0$R::$
       1 == 1$R::$
       constant? p ==  p case R
       constant p ==
          p case R => p
          p.c0
       ...
```

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

4.3.2 Scratchpad vous rend la vie simple:

Le constructeur `XRPOLY` demande qu'on lui passe en paramètre l'alphabet `VarSet` contenant la liste des variables utilisables, ce qui n'est pas toujours très pratique, surtout si l'on désire employer des noms de variables quelconques. Pour pallier à cette contrainte, on définit 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`[4] appartenant à la catégorie des `OrderedSet`.

[3] L'alphabet X est noté `VarSet`

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

XPolynomial

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

4.4 Schéma récapitulatif

Nous donnons dans la figure 2, une architecture de l'implantation des polynômes non commutatifs.

4.5 Algorithmes

La représentation distribuée est souvent pénalisante car le nombre maximum de monômes d'un polynôme non commutatif croît de façon exponentielle par rapport à son degré.

Exemple 4.1 *Soit X un alphabet de 2 lettres; le nombre de mots de longueur d est 2^d. Ce nombre serait de $d+1$ en algèbre commutative.*

4.5.1 Calcul du shuffle en représentation distribuée

Le shuffle de deux polynômes s'obtient à partir du shuffle sur les mots. Soient

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

On a alors:

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

4.5.2 Calcul du shuffle en représentation récursive

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

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

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

- son résiduel pour chaque lettre de l'alphabet:

$$(p \sqcup\!\sqcup q) \triangleright z = (p \triangleright z) \sqcup\!\sqcup q + p \sqcup\!\sqcup (q \triangleright z)$$

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

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

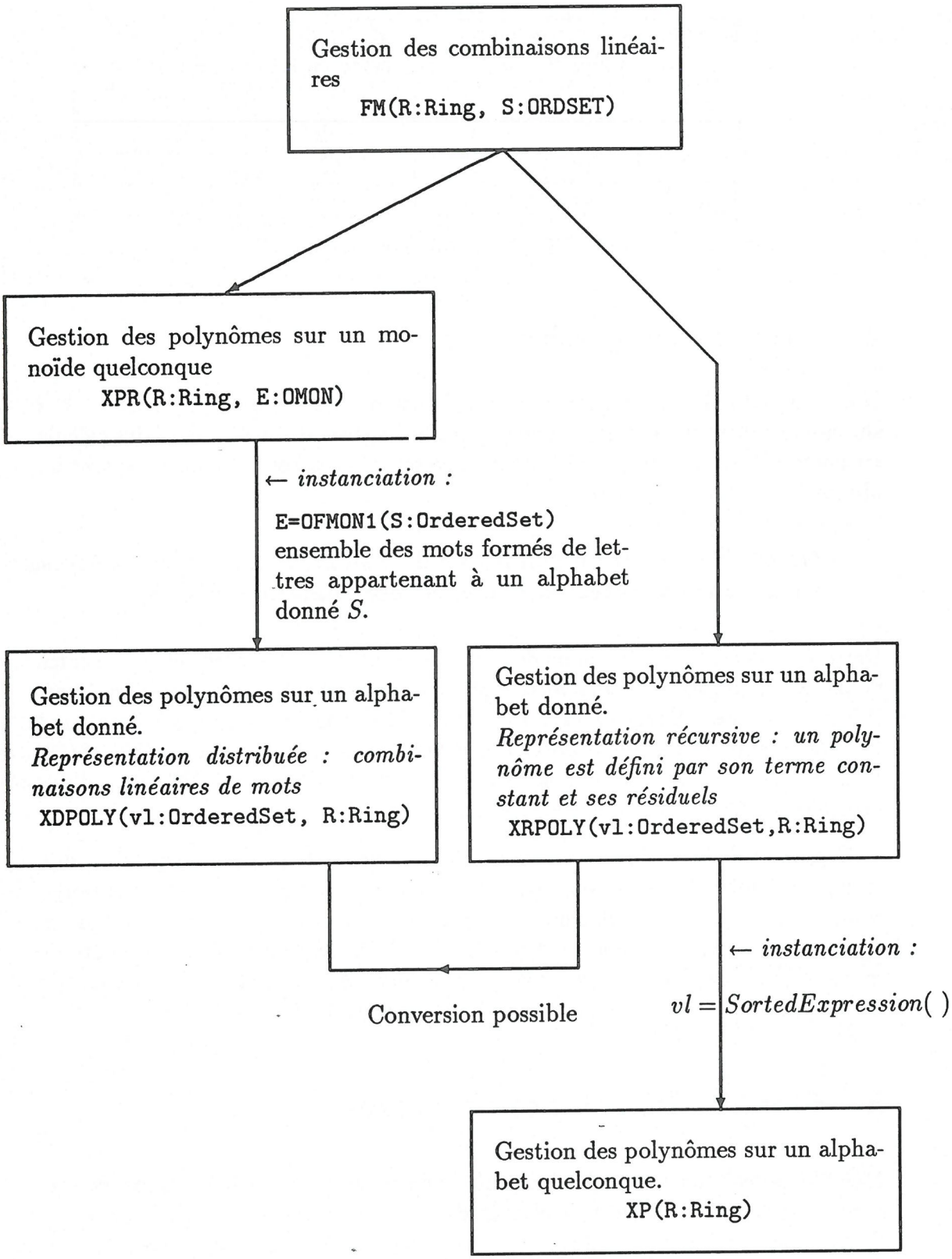

Figure 2: Architecture de l'implantation des polynômes non commutatifs

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

Table 1: Tableau comparatif

4.6 Test et comparaison

Nous donnons dans l'annexe 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 varie dans un facteur 30 selon la représentation choisie.**

Dans le tableau 1 suivant, nous donnons des statistiques sur les temps de calcul pour les deux représentations (récursive et distribuée). Nous ne donnons pas les temps pour les mêmes calculs en *Macsyma* car, d'une part, *Macsyma* est interprété alors que SCRATCHPAD est compilé et d'autre part, *Macsyma* tourne sur une station SUN 3/80 avec 8Mo de mémoire centrale alors que SCRATCHPAD est installé sur un IBM PCRT d'une mémoire centrale de 16Mo et une architecture RISC.

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...

5 Conclusion et perspectives

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. Elles synthétisent toute l'expérience du génie logiciel (programmation orientée objet, types abstraits de données). Pour caricaturer un peu, en *Scratchpad,* on n'implante pas des algorithmes, **on construit des objets informatiques réutilisables par d'autres.**

En ce qui nous concerne, la tâche en algèbre non commutative est immense. Ici, nous avons implanté les polynômes non commutatif et nous avons montré le rôle joué par la

représentation interne dans l'efficacité des algorithmes.

Ce travail n'est que le premier pas. Nous nous en servons déjà dans l'implantation de l'algorithme de la réalisation minimale (G.Jacob, N.E.Oussous et M.Petitot [9]) et nous donne un espoir de pouvoir :

- Implanter les séries rationnelles,
- Faire des calculs dans les algèbres de Lie,
- Faire des applications diverses.

References

[1] B.W.Char *et al.*.– A Tutorial Introduction to Maple, *J. Symbolic Computation* **2**, p. 179-200, 1986.

[2] J.Fitch.– Solving algebraic problems with Reduce, *J. Symbolic Computation* **1**, p. 211-227, 1985.

[3] M.Fliess.– Fonctionnelles causales non linéaires et indéterminées non commutatives, *Bull. Soc. Math. France,* 109, p.3-40, 1981.

[4] G.Jacob and N.Oussous.– Sur un résultat de REE: séries de Lie et algèbres de mélange, Publication du LIFL Lille, IT–103, 1987.

[5] G.Jacob and N.Oussous.– Algebraic computation of analytical minimal realisation, *Publication du LIFL Lille,* IT–139, 1988.

[6] G.Jacob and N.Oussous.– Local and minimal realization of nonlinear dynamical systems and Lyndon words, *IFAC Symposium "Nonlinear Control Systems Design", Capri–Italy,* June 1989.

[7] N.E.Oussous.– Macsyma Computation of Local Minimal Realization of Dynamical Systems of which Generating Power Series are Finite, *to appear in J. Symbolic Computation,* 1990.

[8] N.E.Oussous.– Computation, on Macsyma, of the minimal differential representation of noncommutative polynomials, *to appear in "Algebraic and Computing Traitment of Noncommutative power Series" (ed. G.Jacob and C.Reutenauer), Theoret. Comput. Sci.,* 1990.

[9] N.Oussous et M.Petitot.– Scratchpad implementation of the local minimal realization of dynamic systems, *"Algebraic Computation in Control", Paris–France,* Mars 1991.

[10] R.Pavelle, P.S.Wang.– Macsyma from F to G, *J. Symbolic Computation* **1**, p. 69-100, 1985.

Annexe

```
-- fichier de test  XRPOLY  : polynomes non commutatifs (recursifs)
--                  XDPOLY  : polynomes non commutatifs (distribues)
-- comparaison des resultats (calculs de shuffle)
-------------------------------------------------------------------------
)load XRPOLY XDPOLY )cond      -- initialisation
)time on
)clear all

alph:= OV [z,y,x]              -- definition des domaines
mot:=OFMON1 alph
rpoly:=XRPOLY(alph,I)
dpoly:=XDPOLY(alph,I)

x:alph:=x                      -- forcage de type sur les variables
y:alph:=y
z:alph:=z

pr :rpoly:=2*x+3*y+5           -- polynomes de degre 1
pd :dpoly:= extend(pr)

pr2 := pr*pr                   -- calcul de la puissance 2
pd2 := pd*pd
extend(pr2) =$dpoly pd2        -- egalite des resultats?

pr6:rpoly:=pr2*pr2*pr2         -- calcul de la puissance 6
pd6:dpoly:=pd2*pd2*pd2
extend(pr6) - pd6              -- egalite des resultats?

qr:= sh(pr6,pr2)               -- calcul du shuffle
qd:= sh(pd6,pd2)
extend(qr)-qd                  -- egalite des resultats?
```

Voici une trace de l'exécution des calculs:

```
;alph:= OV [z,y,x]            -- definition des domaines
   (1)  OV [z,y,x]
   Type: DOMAIN                                        .2 (IN) = .2 sec
........
;x:alph:=x                    -- forcage de type sur les variables

   (5)  x
   Type: OV [z,y,x]                                              0 sec
-------------------------------------------------------------------------
```

```
;pr :rpoly:=2*x+3*y+5          -- polynomes de degre 1

   (8)  5 + x 2 + y 3
   Type: XRPOLY(OV [z,y,x],I     .917 (IN) + .1 (EV) + .1 (OT) = 1.117 sec


;pd :dpoly:= extend(pr)

   (9)  5 + 2x + 3y
   Type: XDPOLY(OV [z,y,x],I                                  .1 (OT) = .1 sec

;pr2 := pr*pr                   -- calcul de la puissance 2

   (10)  25 + x(20 + x 4 + y 6) + y(30 + x 6 + y 9)

   Type: XRPOLY(OV [z,y,x],I                        .3 (IN) + .1 (EV) = .4 sec

;pd2 :=pd*pd
                              2                      2
   (11)  25 + 20x + 30y + 4x  + 6x y + 6y x + 9y
   Type: XDPOLY(OV [z,y,x],I                   .367 (IN) + .033 (EV) = .4 sec

;extend(pr2) =$dpoly pd2       -- egalite des resultats?

   (12)  true
   Type: B                        .1 (IN) + .2 (EV) + .033 (OT) = .333 sec

;pr6:rpoly:=pr2*pr2*pr2        -- calcul de la puissance 6
   ....
   Type: XRPOLY(OV [z,y,x],I         .5 (IN) + .1 (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
     45000y x
   ....  resultat sur 1 page
   +
         2 2 2        2     2       2                2   2         2   3
     324y x y  + 216y x y x  + 324y x y x y + 324y x y x + 486y x y
   +
```

```
        3 3       3 2         3             3   2        4 2        4
     216y x + 324y x y + 324y x y x + 486y x y  + 324y x  + 486y x y
   +
        5         6
     486y x + 729y
   Type: XDPOLY(OV [z,y,x],I)
                              1.533 (IN) + .967 (EV) + .033 (OT) = 2.533 sec

;extend(pr6) - pd6              -- egalite des resultats?
   (15)  0
   Type: XDPOLY(OV [z,y,x],I .2 (IN) + 1.267 (EV) + .167 (OT) = 1.633 sec

;qr:= sh(pr6,pr2)               -- calcul du shuffle
   (16)
   ....
   Type: XRPOLY(OV [z,y,x],I).1 (IN) + 15.233 (EV) + .2 (OT) = 15.533 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
   +

   ...   resultats sur 5 pages

              3             2               2 2                2
     122472y x y  + 54432y x  + 81648y x y + 81648y x y x + 122472y x y
   +
           6 2           6             7           8
     81648y x  + 122472y x y + 122472y x + 183708y

   Type: XDPOLY(OV [z,y,x],I)
                         .1 (IN) + 263.667 (EV) + 33.833 (OT) = 297.601 sec

;extend(qr)-qd                  -- egalite des resultats?

   (18)  0
   Type: XDPOLY(OV [z,y,x],I)                          2.767 (EV) = 2.767 sec
```