UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

LABORATOIRE D'INFORMATIQUE DE PARIS 6

MÉMOIRE D'HABILITATION À DIRIGER DES RECHERCHES

Présenté par

RENAUD RIOBOO

le 18 décembre 2002
en présence du jury composé de

René Alt, Président
James Davenport, rapporteur
Gilles Kahn, rapporteur
Tomas Recio, rapporteur
Thérèse Hardin, directrice
Daniel Lazard, examinateur
Xavier Leroy, examinateur

Remerciements

Réunir un jury fin décembre dans un campus en cours de désamiantage n'est pas très simple.

René Alt a bien voulu accepter de présider ce jury, j'en suis très honoré et je l'en remercie vivement.

James Davenport a bien voulu me faire l'honneur de rapporter sur ces travaux. Ses écrits, ses réflexions et son exemple m'ont toujours guidé dans ma recherche. Je le remercie d'avoir trouvé le temps de rapporter sur ce document.

Gilles Kahn a toujours manifesté de l'intérêt pour le calcul formel qu'il a beaucoup soutenu. On ne peut pas aborder la sémantique ou la validation de programmes sans avoir son exemple à l'esprit. Je le remercie ici de rapporter sur mes travaux.

Tomas Recio a bien voulu apporter sa vision de mathématicien sur mes travaux. Je le remercie d'avoir accepté de rapporter sur ce document dont une partie est assez éloignée de la géométrie algébrique dont il a l'habitude.

Le calcul formel en France ne serait rien sans Daniel Lazard. C'est lui qui m'a appris tout ce que je sais en calcul formel. Je le remercie de participer à ce jury.

Xavier Leroy a accepté de faire partie de ce jury. Je le remercie aussi pour le merveilleux langage qu'il nous permet d'utiliser et sans lequel le projet FoC n'aurait pas pu exister.

Enfin, Thérèse Hardin a accepté de m'accompagner sur la voie périlleuse de la certification d'algorithmes de calcul formel. Depuis 5 ans maintenant elle dirige le projet FoC, je la remercie ici pour le travail impressionnant qu'elle a fourni. Cette habilitation n'aurait probablement pas lieu sans elle.

La conception de nouveaux paradigmes de programmation ne peut se faire de manière isolée. Mon travail doit beaucoup aux différents stagiaires ou thésards ainsi qu'aux autres membres du projet FoC. La liste serait trop longue à faire ici, qu'ils sachent bien que leur travail et leur idées m'ont beaucoup aidé.

Au-delà des membres du projet, mon travail a bénéficié des conseils avisés et de l'aide de tous les membres des thèmes Calfor et Spi. Beaucoup de discussions avec les membres des projets Inria Lemme, Cristal ou Coq m'ont également permis de mieux aborder la spécificité de la sémantique dans le cadre du calcul formel. Qu'ils en soient ici remerciés.

Table des matières

Introduction **4**

I Les nombres algébriques réels **8**

1 Le cadre mathématique **11**
1.1 Rappels sur les polynômes en une variable 11
1.1.1 Définitions 11
1.1.2 Propriétés 12
1.2 Le résultant 14
1.2.1 Définitions 14
1.2.2 Propriétés élémentaires 17
1.3 Corps et extensions 18
1.3.1 Éléments primitifs 19
1.3.2 Corps réels clos 21
1.4 Algorithmes de base 22
1.4.1 Suite de Sturm 22
1.4.2 Comptage des racines 24
1.4.3 Calcul de signes 25

2 La clôture réelle **27**
2.1 Les tours d'extensions 28
2.2 Un modèle abstrait 29
2.2.1 Opérations Génériques 31
2.2.2 Un modèle en FoC 32
2.2.3 Le codage par intervalles 32
2.3 Extensions de corps ordonnés 34
2.3.1 Les extensions simples 34
2.3.2 Les extensions multiples 36
2.3.3 Un exemple en FoC 37

3 Les sous résultants **42**
3.1 Les sous résultants 42
3.2 Les sous résultants faibles 44
3.3 Un modèle d'anneaux 46
3.4 Les suites de Sturm faibles 47

Conclusions **49**

II FOC **51**

4 Motivations **54**
4.1 Un exemple 55
4.2 Vers une maîtrise de la sémantique 56
4.3 Quelques traits des langages de programmation 57
4.3.1 La programmation fonctionnelle 57
4.3.2 La surcharge 59
4.3.3 Le typage 60
4.3.4 L'héritage 63
4.3.5 La liaison tardive 64

5 Le modèle FoC **66**
5.1 Le codage 66
5.1.1 Méthodologie 67
5.1.2 Approche par module 68
5.1.3 Approche naïve 72
5.1.4 Approche par représentation 77
5.1.5 Un bilan 83
5.2 Notations 85
5.3 Types et valeurs 87
5.3.1 Types atomiques 87
5.3.2 Constructions 87
5.3.3 Paramètres 88
5.3.4 Polymorphisme 89
5.3.5 Comparaison avec Axiom 89
5.4 Espèces et Collections 90
5.4.1 Les composantes 91
5.4.2 Le gel 93
5.4.3 Paramétrage 94
5.4.4 L'héritage 95
5.4.5 Surcharge 96

5.4.6 Comparaison avec Axiom 96

6 Implémentation des polynômes en FoC **99**
6.1 Représentations distribuées 100
6.2 Représentations récursives 101
6.3 Généralisation 104

Conclusions **105**

Conclusion générale **110**

Bibliographie **114**

Introduction

Ce document présente mes travaux depuis ma thèse. Ils portent pour une part sur le développement de nouveaux algorithmes pour le calcul exact sur les nombres réels. L'autre part traite de la construction d'un environnement de programmation certifiée adapté au calcul formel.

Pour calculer efficacement avec des nombres algébriques, nous verrons dans la première partie que nous serons amenés à en avoir une « vision récursive ». L'implantation de ces algorithmes nous a amené à nous poser la question du « sens» des constructions syntaxiques du langage de programmation que nous utilisons quand nous écrivons nos codes. Plus généralement, mon expérience de programmation dans divers systèmes m'a conduit à une réflexion sur le langage de programmation « idéal » pour implanter des algorithmes de calcul formel.

En tant que programmeurs de calcul formel nous manipulons des expressions algébriques. Leur sens est celui des mathématiques sous-jacentes. Pour nous le langage de programmation est un générateur de formules. Nous avons ainsi tendance à voir les expressions du langage comme des expressions mathématiques. Nous voyons les constructions syntaxiques que nous utilisons comme des méta-formules. Nous leur demandons souvent la même concision que celle des expressions algébriques que nous manipulons. Autrement dit, nous avons une vision mathématique de la syntaxe de notre langage de programmation.

Un système comme MAPLE en est un bon exemple, tout y est mathématique, c'est une calculette pour amateurs d'expressions algébriques. De tels systèmes considèrent toutes les données comme des « objets mathématiques » sur lesquels on peut appliquer toutes les opérations algébriques. Le système « lit » des expressions, les « simplifie » en d'autres expressions qu'il « affiche »[1]. C'est toujours le lecteur qui détermine leur « sens » à l'aide de son « intuition » mathématique.

Malheureusement, on ne peut pas décrire de manière satisfaisante des

[1] n'oublions pas que les premiers systèmes de calcul formel comme MACSYMA ou REDUCE sont issus de LISP.

algorithmes compliqués dans un tel système. En tant que programmeur de calcul formel, on est amené à faire des hypothèses sur les données au fur et à mesure des calculs. Par exemple on peut être amené à ne travailler qu'avec des polynômes en X à coefficients sur les entiers. On voudrait pouvoir dire que la procédure que l'on est en train d'écrire travaille sur ces polynômes et uniquement ceux ci.

Ces contraintes sur les données sont difficiles à décrire en MAPLE qui ne gère que des « objets mathématiques ». Il y est difficile de faire des « déclarations », c'est le programmeur qui a l'entière responsabilité des données qu'il manipule. Il dispose de certaines aides, mais en fin de compte c'est lui qui doit tester si la donnée que reçoit sa procédure est bien un polynôme en X à coefficients entiers.

Dans une calculette formelle, le mathématicien ne peut pas « parler » de mathématiques, il ne peut que les « faire ». Le chercheur en calcul formel s'attendrait à pouvoir formuler une vision plus descriptive des ses algorithmes : « la technique ne s'applique qu'aux polynômes à coefficients entiers ». Cette phrase courante dans les laboratoires où on est amené à faire du calcul formel conduit à ne pas se limiter à l'aspect calculatoire du système. On voudrait pouvoir ajouter une déclaration qui garantisse cette propriété.

Toutefois, l'ajout de composantes descriptives (déclarations, propriétés ...) freine leur utilisation comme « calculette ». Certains utilisateurs, des mathématiciens par exemple, connaissent « très bien » la signification de leurs formules et souhaitent pas avoir à l'expliciter dans le système. Par exemple un mathématicien connaît les propriétés d'un résultant, il ne peut simplement pas faire le calcul (trop compliqué) à la main. Il peut le manipuler sans même le « lire ». D'autres utilisateurs ne s'intéressent pas au cadre mathématique de leur problème. Ainsi un physicien se place naturellement dans le cadre des nombres réels ou complexes et attend du système une aide « naturelle » pour manipuler ses formules.

On a donc un antagonisme clair entre l'utilisateur de « programmes mathématiques » et le développeur de programmes (mathématiques bien sûr). Comme dans n'importe quel développement de logiciel, l'utilisateur veut de la souplesse que le programmeur ne peut lui accorder. En calcul formel, le développeur est souvent utilisateur. En tant que programmeur il veut de la rigueur et en tant qu'utilisateur il veut des outils faciles à utiliser. J'entendais récemment l'expression « shell mathématique ».

Ajouter des composantes descriptives au langage de programmation amène l'implémenteur de calcul formel à abandonner sa vison « méta-formule » de la syntaxe qu'il utilise. Les expressions mathématiques qu'il manipule doivent se « concevoir » en tant qu'élément d'ensemble sur lesquels on applique des opérations. On est ainsi plus « près » de leur véritable « nature» mathématique.

Les concepteurs de systèmes comme SCRATCHPAD/AXIOM puis ALDOR ou bien MAGMA ont toujours considéré que les composantes descriptives étaient nécessaires. Ils se placent donc du point de vue du programmeur ou du mathématicien qui fait d'abord une théorie avant de l'appliquer. L'utilisateur ce ces systèmes doit « entrer » dans le modèle et accepter de décrire les objets avant de les manipuler. Le système peut les aider en affichant la « nature » de la donnée en même temps que sa valeur.

De tels systèmes permettent souvent une meilleure efficacité pratique. Plus un objet mathématique est « décrit », plus son « codage » peut être « simple ». Les entiers en base 10 sont « plus petits » que les entiers de Peano parce qu'on a induit une partie des propriétés de l'arithmétique dans leur codage. La représentation des objets est plus petite et on a de bons algorithmes pour la manipuler.

C'est ce point de vue de développeur que j'ai toujours adopté. On ne fait aucune hypothèse sur ce que doit être un « système » de calcul formel et on veut simplement un « langage», aussi riche que possible pour implanter des algorithmes de calcul formel.

Dans ce document nous parlerons donc d'algorithmes et de langages de programmation. Pour décrire les algorithmes nous utiliserons souvent un pseudo AXIOM où les mots clefs et les identificateurs ont une forme mathématique. Ce pseudo langage utilise les conventions d'AXIOM :

- les instructions sont séquencées sur des lignes indentées de la même façon. Une séquence retourne la valeur de la dernière expression évaluée.
- L'affectation se note « $\leftarrow$ ».
- La notation « $\Rightarrow$ » sert de structure de contrôle, ainsi

  test $\Rightarrow$ result
  continue

  signifie que si « test » est vrai alors on évalue « result » sinon on passe à l'instruction suivante. C'est a dire que l'on évalue « continue ». Lorsque « result » nécessite l'évaluation de plusieurs expressions on les mettra sur de nouvelles lignes en les indentant par rapport à « continue ».
- Les fonctions sont implicitement récursives.
- Lorsque c'est possible nous nous autoriserons à noter les identificateurs comme en mathématiques. Ainsi si on utilise un nom « `x` » on pourra le noter « x » si on ne risque pas de faire la confusion entre le nom « `x` » et la valeur « x » qu'il prend.
- Une ligne commençant par un double signe « `-` » ou un double signe « `+` » désigne une commentaire.

Nous utiliserons parfois directement la syntaxe d'AXIOM décrite dans [BBD+]. En général j'essaierai de ne pas mettre trop de code. Nous utiliserons souvent la syntaxe d'OCAML qui est décrite dans [LDG+00]. La syntaxe de FOC s'en inspire, elle est décrite dans la section 5.2.

Ce document est composé de deux parties. La première présente les algorithmes de calcul exact sur les nombres réels que je propose. J'ai choisi d'en faire une présentation la plus intuitive possible, quitte à sacrifier parfois la rigueur mathématique. Les articles annexés à ce document donnent les formalismes mathématiques complets et les preuves des algorithmes.

Dans la seconde partie, j'essaierai de faire état des réflexions, expériences ou intuitions qui ont conduit à la conception de FOC. Le système FOC est un environnement pour la programmation certifiée d'algorithmes de calcul formel. Il propose un langage source qui est compilé à la fois vers un programme exécutable et vers un système d'aide à la preuve. Le langage possède une librairie de calcul formel. La présentation de ce document met l'accent sur l'aspect programmation du langage et sur la librairie qui l'utilise.

Première partie

Les nombres algébriques réels

Les nombres algébriques sont, par définition, les solutions des équations polynômiales en une variable. Ainsi 1 est un nombre algébrique, $\sqrt{2}$ ou les racines du polynôme $X^5 - X + 1$ aussi. Ils interviennent naturellement en calcul formel car c'est en termes de nombres algébriques que s'expriment les solutions d'un système d'équations polynômiales à coefficients entiers. Ainsi beaucoup d'algorithmes de résolution de systèmes d'équations considèrent un corps de base algébriquement clos. Les solutions réelles d'un système s'expriment elles naturellement dans un corps réel clos.

Le calcul effectif dans un corps réel clos nécessite de pouvoir extraire et manipuler des racines de polynômes en une variable. Il y a de bons algorithmes pour les extraire lorsque les coefficients des polynômes sont des flottants. Les racines obtenues restent des flottants où les opérations de base s'effectuent en temps constant. Elles ne sont donc pas plus « compliquées » que les coefficients. Ainsi, pour le programmeur FORTRAN, une racine carrée n'est pas plus compliquée que le nombre de départ. Malheureusement, l'arithmétique sur les flottants n'a pas les propriétés nécessaires pour être utilisée directement par le calcul formel. Nous utilisons donc souvent des arithmétiques *exactes* qui sont coûteuses parce que la taille des objets peut être arbitrairement grande.

Une arithmétique exacte est une arithmétique dans laquelle on peut calculer « comme en mathématiques », on doit donc implémenter les 4 opérations arithmétiques de base et pouvoir tester si deux éléments sont égaux. Pour la clôture réelle qui nous intéresse ici, on veut en plus pouvoir décider si un nombre est plus petit qu'un autre.

Résoudre un système d'équations revient souvent à donner les nombres algébriques qui sont solutions de ce système quand ils existent. On laisse ensuite à l'arithmétique des nombres algébriques le soin de les manipuler. C'est par exemple le cas pour le « solver » d'AXIOM qui produit des « solutions » qui sont ensuite manipulée grâce à l'arithmétique des réels algébriques.

Il suffit d'ouvrir un livre sur la théorie des corps pour s'apercevoir que la manipulation de nombres algébriques est compliquée. Les calculs mettant en œuvre des nombres algébriques sont, en pratique, difficiles pour le calcul formel où l'arithmétique doit être exacte. Ainsi beaucoup de questions sur les nombres algébriques sont encore ouvertes et leur interprétation « naturelle » reste difficile. Par exemple [DST87] propose un certain nombre d'équations faisant intervenir des nombres algébriques. Ils sont tous mal ou pas résolus par les systèmes de calcul formel. Comme on peut produire des exemples aussi compliqués que l'on veut on a une source de « challenges » pour l'arithmétique algébrique. Les équations de [DST87] n'ont été résolus de manière satisfaisante que grâce à la clôture réelle (voir la section 2) que j'ai développée.

Un autre intérêt des nombres algébriques réels est qu'il peuvent être utilisés « naïvement » et servir de « brique » pour d'autres algorithmes. Par exemple la décomposition cylindrique de Collins utilise une arithmétique réelle close. La résolution de système à base d'ensembles triangulaires produit des « solutions » qui sont des listes de polynômes dont le premier est en une variable, le second en deux et ainsi de suite. Pour trouver les points qui sont solution de cet ensemble triangulaire une méthode naturelle est de prendre les racines du premier polynôme. On évalue ensuite les autres polynômes pour une racine donnée du premier polynôme, ce qui nous ramène au même problème avec une variable de moins. On peut ainsi « trouver » toutes les solutions.

Dans une application, comme la décomposition cylindrique ou la résolution de systèmes, on effectue beaucoup de calculs avec des nombres algébriques réels. On s'apperçoit rapidement que la plupart des calculs sont très « gourmands » en arithmétique rationnelle. Ainsi la majorité du temps de calcul est passée à simplifier des rationnels. C'est ce qui a motivé ma réflexion sur des algorithmes spécifiques pour les polynômes dont les coefficients sont des nombres algébriques.

Nous donnerons d'abord le formalisme mathématique nécessaire pour travailler avec des nombres algébriques. Nous présenterons ensuite le modèle et certains algorithmes utilisés dans le code AXIOM implémentant les nombres algébriques réels. Le chapitre 3 est consacré aux améliorations que j'ai proposées récemment (voir [Rio02]).

Chapitre 1

Le cadre mathématique

Nous rappelons ici les outils nécessaires pour pouvoir travailler effectivement avec des nombres algébriques réels. Nous rappelons d'abord quelques définitions de base sur les polynômes en une variable. Nous donnons ensuite quelques propriétés des résultants qui permettent de mieux aborder les extensions de corps.

1.1 Rappels sur les polynômes en une variable

1.1.1 Définitions

Si $\mathbf{A}$ est un anneau commutatif et G est un monoide multiplicatif dont l'élément neutre est noté 1, Lang ([Lan69]) définit les polynômes $\mathbf{A}[G]$ comme l'ensemble des fonctions de G dans $\mathbf{A}$ qui sont nulles presque partout, c'est à dire sauf en nombre fini de points. Si P et Q sont dans $\mathbf{A}[G]$ l'addition des polynômes est l'addition des fonctions et la multiplication est donnée par la fonction PQ qui en un point c de G vaut

$$\sum_{a.b=c} P(a)Q(b)$$

Pour $a \in \mathbf{A}$ et $x \in G$ on note $a.x$ la fonction qui vaut a au point x et 0 partout ailleurs. Un élément quelconque de $\mathbf{A}[G]$ peut se mettre sous la forme d'une somme $\sum_{x \in G} a_x x$ dont presque tous les termes sont nuls. Le polynôme nul correspond à la fonction qui vaut 0 en tout point de G.

$\mathbf{A}[G]$ est un anneau, on en fait un $\mathbf{A}$-module en prolongeant la multiplication terme à terme. On obtient ainsi la $(\mathbf{A}, G)$-algèbre libre dont une base est $\{x\}_{x \in G}$ en identifiant x et $1.x$.

Définition 1 (Polynômes univariés)
Les polynômes en une variable X sur un anneau commutatif $\mathbf{A}$ s'obtiennent en prenant comme monoide multiplicatif le monoide libre $\{X\}^*$ dans la construction précédente. Par convention on note $\mathbf{A}[X]$ plutôt que $\mathbf{A}[\{X\}^*]$.

Un élément $X \ldots X$ de $\{X\}^*$ contenant i occurrences de X se note X^i. On écrit un polynôme en une variable sous la forme

$$\sum_i a_i X^i$$

en ne faisant figurer que les termes non nuls. Le polynôme nul se note 0. Un élément de la forme X^i est un monôme primitif[1] et l'indice i est son *degré*. Un élément de la forme $a_i X^i$ est un monôme[2] de degré i et a_i est son *coefficient*.

Pour un polynôme P non nul[3] on appelle *monôme dominant* d'un polynôme le monôme de P de plus grand degré. On appelle *degré* d'un polynôme et on note $\mid P \mid$ le degré de son monôme dominant. On prend parfois 0 ou -1 comme degré du polynôme nul. Le *coefficient dominant* d'un polynôme P est le coefficient de son monôme dominant que l'on note $\mathrm{lc}(P)$. On prend souvent 0 comme coefficient dominant du polynôme nul.

1.1.2 Propriétés

Lorsque $\mathbf{A}$ est un corps, $\mathbf{A}[X]$ est un anneau euclidien c'est à dire muni une division euclidienne. L'algorithme d'Euclide permet de calculer un plus grand facteur commun à deux polynômes P et Q de $\mathbf{A}[X]$.

Lorsque deux polynômes P et Q sont premiers entre eux, on peut trouver deux polynômes B_P et B_Q pour obtenir une relation de Bezout :

$$B_P P + B_Q Q = 1$$

On peut étendre l'algorithme d'Euclide pour calculer les coefficients de Bezout B_P et B_Q en même temps que le pgcd de deux polynômes P et Q de la manière suivante :

$\mathtt{bezout}(P, Q) =$
$\quad Q = 0 \Rightarrow (1, 0, P)$
$\quad (K, R) \leftarrow \mathtt{quo\_rem}(P, Q)$
$\quad --$ on a donc$P = KQ + R$
$\quad (B_Q, B_R, G) \leftarrow \mathtt{bezout}(Q, R)$
$\quad B_P \leftarrow B_R$
$\quad B'_Q \leftarrow B_Q - KB_R$
$\quad (B_P, B'_Q, G)$

[1] parfois appelé simplement monôme
[2] parfois appelé terme
[3] le polynôme nul n'a pas de monômes

On obtient ainsi une relation :

$$B_P P + B_Q Q = G$$

qui exprime effectivement le pgcd G de P et de Q comme combinaison linéaire de P et de Q.

Comme la division euclidienne nécessite d'inverser le coefficient dominant d'un polynôme ces relations ne sont valables que lorsque $\mathbf{A}$ est un corps. En général, on se place sous des hypothèses plus faibles. L'anneau $\mathbf{A}$ de base est simplement un anneau intègre et on ne peut inverser son coefficient dominant.

On a toutefois une notion de *pseudo division* qui peut s'écrire de la manière suivante :

$$\forall P, Q \in \mathbf{A}[X],\ \exists K, R \in \mathbf{A}[X],\ \ \mathrm{lc}(Q)^{\max(\delta+1,0)} P = KQ + R$$

où δ est la différence des degrés de P et de Q. R se nomme le *pseudo reste* et Q se nomme le *pseudo quotient* de la pseudo division. Cette décomposition est unique et en notant $\mathrm{rem}(P, Q)$ le reste de la division euclidienne de P par Q dans le corps des fractions de $\mathbf{A}$, le pseudo reste $\mathrm{prem}(P, Q)$ vérifie la relation

$$\mathrm{prem}(P, Q) = \mathrm{lc}(Q)^{\max(\delta+1,0)} \mathrm{rem}(P, Q)$$

dans le corps des fractions de $\mathbf{A}$. L'intérêt de la pseudo division vient de ce qu'elle se calcule sans faire de divisions dans $\mathbf{A}$.

Définition 2
On dit qu'un polynôme $P(X)$ de $\mathbf{A}[X]$ est *sans facteurs multiples* ou *sans facteurs carrés* lorsqu'il n'existe pas de diviseur $D(X)$ de $P(X)$ tel que $D^2(X)$ divise P.

D'un point de vue calculatoire, cela revient à dire que $P(X)$ et sa dérivée $P'(X)$ n'ont pas de facteurs communs. En effet, plaçons nous dans une clôture algébrique $\mathbf{F}$ de $\mathbf{A}$, on peut décomposer P en facteurs irréductibles :

$$P = \prod_{i \in \mathcal{I}} (X - \alpha_i)^{\nu_i}$$

où les α_i sont les différentes racines de P et ν_i leur multiplicité. On dérive formellement cette relation pour obtenir

$$P' = \sum_{i \in \mathcal{I}} \left\{ (X - \alpha_i)^{\nu_i - 1} \prod_{j \neq i} (X - \alpha_j)^{\nu_j} \right\}$$

qu'il suffit ensuite d'évaluer en α_i pour obtenir

$$\prod_{j\neq i}(\alpha_i - \alpha_j)^{\nu_j}$$

qui est non nul puisque les α_i sont supposés distincts. Le polynôme P n'a donc que des racines simples :

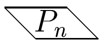
$$P = \prod_{i\in\mathcal{I}}(X - \alpha_i)$$

1.2 Le résultant

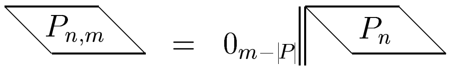
Soit $\mathbf{A}$ un anneau commutatif, $P(X)$ et $Q(X)$ deux polynômes de $\mathbf{A}[X]$. On choisira Q non nul et P de degré supérieur ou égal à celui de Q.

1.2.1 Définitions

Pour $n > 0$ et $m \geq \ \mid P \mid$ on notera $\Delta(P)_{n,m}$ la matrice ayant n lignes et $n+m$ colonnes dont les lignes sont les $n+m$ coefficients (éventuellement nuls) de $X^{n-1}P$, $\ldots P$. Pour $m >\mid P \mid$ la matrice $\Delta(P)_{n,m}$ à donc ses $m- \mid P \mid$ premières colonnes nulles. On abrégera $\Delta(P)_{n,|P|}$ en $\Delta(P)_n$. Schématiquement $\Delta(P)_n$ est donc de la forme :

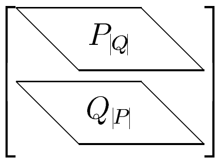


et $\Delta(P)_{n,m}$ est formée de $m- \mid P \mid$ colonnes de zéros devant $\Delta(P)_n$, ce que nous écrivons schématiquement



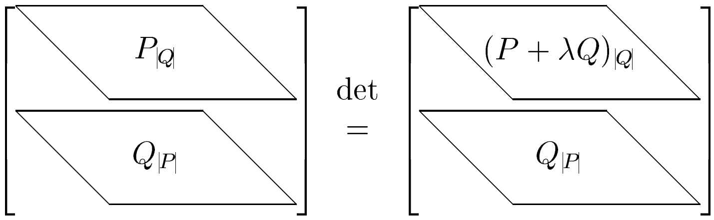
Définition 3 (Résultant)
La *matrice de Sylvester* de P et de Q est formée des lignes de $\Delta(P)_{|Q|}$ suivies de celles de$\Delta(Q)_{|P|}$. On écrit schématiquement :

$$\Delta(P,Q) = \begin{bmatrix} \Delta(P)_{|Q|} \\ \Delta(Q)_{|P|} \end{bmatrix} =$$



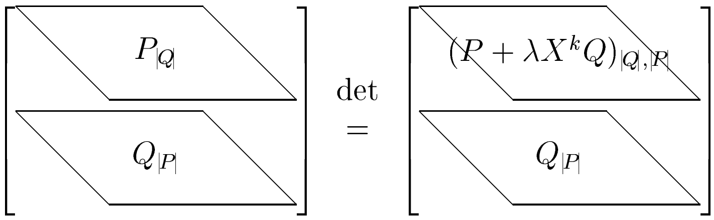
Le *résultant* $\mathrm{Res}(P,Q)$ de P et Q est par définition le déterminant $\mid \Delta(P,Q) \mid$ de la matrice $\Delta(P,Q)$.

Supposons $|P| \geq |Q|$, on sait que le déterminant de $\Delta(P, Q)$ est inchangé si on ajoute à $\Delta(P)_{|Q|}$ des multiples des lignes de $\Delta(Q)_{|P|}$. On ajoute à $\Delta(P)_{|Q|}$ un multiple λ des $|Q|$ dernières lignes de $\Delta(Q)_{|P|}$ on a :



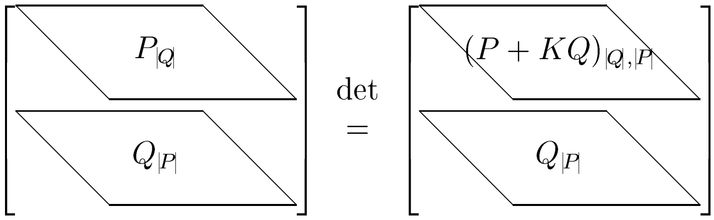
(1.1)

ici la notation $M_1 \overset{\det}{=} M_2$ signifie que les deux matrices M_1 et M_2 ont le même déterminant. En procédant de même avec $|Q|$ lignes de $\Delta(Q)_{|P|}$ de la $(|P| - |Q| - k + 1)$-ème ligne jusqu'à la $(|P| - k + 1)$-ème on voit que



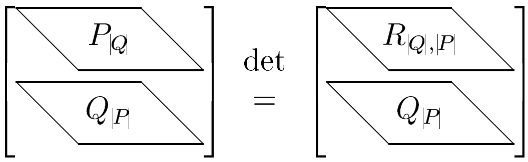
(1.2)

Donc que pour tout polynôme K de degré inférieur ou égal à $|P| - |Q|$ on a



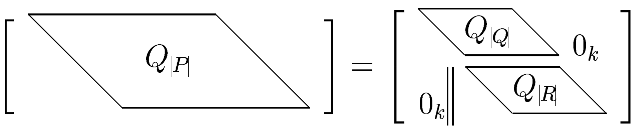
(1.3)

En supposant que l'anneau A est un corps on peut choisir pour K l'opposé du quotient de la division euclidienne de P par Q. On obtient donc :



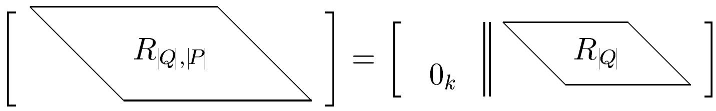
(1.4)

en notant R le reste de la division. Faisons apparaître les $k = |P| - |R|$ premières lignes de $\Delta(Q)_{|P|}$:



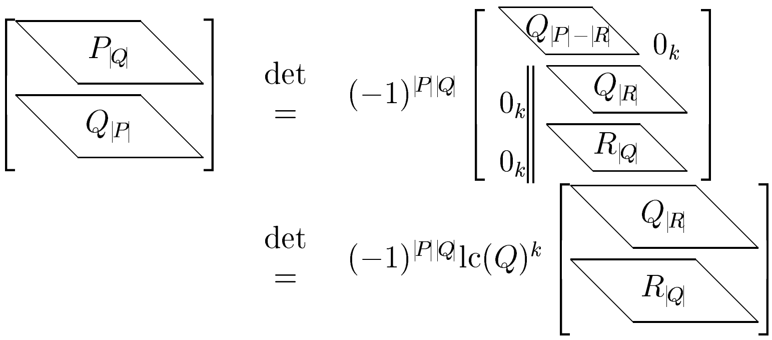
(1.5)

De même faisons apparaître les k premières colonnes de $\Delta(R)_{|Q|}$:



(1.6)

En permutant les lignes de $\Delta(P)_{|Q|}$ et celles de $\Delta(Q)_{|P|}$ chaque ligne de $\Delta(Q)_{|P|}$ remonte de $|\,Q\,|$ positions. sans changer la structure de $\Delta(P)_{|Q|}$ on fait passer la première ligne de $\Delta(Q)_{|P|}$ en $|\,Q\,|$ permutations. Comme $\Delta(Q)_{|P|}$ a $|\,P\,|$ lignes on fait $|\,Q\,||\,P\,|$ permutations pour échanger les lignes de $\Delta(P)_{|Q|}$ et celles de $\Delta(Q)_{|P|}$.



(1.7)

puisque les k premières lignes ont $\mathrm{lc}(Q)$ en position diagonale. Et donc :

$$\mathrm{Res}(P,Q) = (-1)^{|P||Q|}\mathrm{lc}(Q)^{|P|-|R|}\mathrm{Res}(Q,R)$$

que l'on peut prendre comme équation de définition du résultant ce qui conduit à l'algorithme suivant.

Algorithme 1 (Résultant naïf)
D'un point de vue effectif l'algorithme de calcul du résultant de deux polynômes P et Q non nuls s'écrit donc :

$$\begin{array}{l}
\texttt{resultant}(P,Q) = \\
\quad |\,Q\,| = 0 \Rightarrow \mathrm{lc}(Q)^{|P|} \\
\quad R \leftarrow \texttt{rem}(P,Q) \\
\quad (-1)^{|P||Q|}\mathrm{lc}(Q)^{|P|-|R|}\texttt{resultant}(Q,R)
\end{array}$$

en considérant que le degré du polynôme nul est égal à 0. On peut mettre cet algorithme sous forme récursive terminale en ajoutant un paramètre supplémentaire λ initialisé à 1 :

$$\begin{array}{l}
\texttt{resultant}(P,Q,\lambda) = \\
\quad |\,Q\,| = 0 \Rightarrow \lambda\mathrm{lc}(Q)^{|P|} \\
\quad R \leftarrow \texttt{rem}(P,Q) \\
\quad \texttt{resultant}(Q,R,(-1)^{|P||Q|}\mathrm{lc}(Q)^{|P|-|R|}\lambda)
\end{array}$$

en considérant toujours que le degré du polynôme nul est 0.

On voit donc bien que l'algorithme de calcul du résultant est analogue à l'algorithme d'Euclide puisque si on remplace la condition d'arrêt

$$|Q| = 0 \Rightarrow \lambda \mathrm{lc}(Q)^{|P|}$$

par la condition

$$\begin{array}{l} |Q| = 0 \Rightarrow \\ \quad Q = 0 \Rightarrow P \\ \quad 1 \end{array}$$

le paramètre λ ne sert à rien et on obtient bien l'algorithme d'Euclide.

1.2.2 Propriétés élémentaires

En examinant l'algorithme 1 précédent, on voit que le résultant de deux polynômes à coefficients sur un corps est nul si et seulement si ils ont un facteur commun non trivial. En effet on n'atteint la condition $|Q| = 0$ que lorsque Q est nul et alors $\mathrm{lc}(Q)$ vaut 0.

Comme les déterminants de matrices se conservent par homomorphisme d'anneaux, les résultants ont de bonnes propriétés de transfert. On peut ainsi les calculer dans un anneau pour leur donner un sens dans un autre anneau.

Soient deux anneaux $\mathbf{A}$ et $\mathbf{B}$ et soit F un homomorphisme d'anneaux de $\mathbf{A}$ dans $\mathbf{B}$. On étend canoniquement F en un homomorphisme de $\mathbf{A}[X]$ dans $\mathbf{B}[X]$ encore noté F. Soient $P_{\mathbf{A}}$ et $Q_{\mathbf{A}}$ deux polynômes de $\mathbf{A}[X]$ et $P_{\mathbf{B}}$ et $Q_{\mathbf{B}}$ leurs images par F dans $\mathbf{B}[X]$. Si les degrés des polynômes sont conservés par F les matrices de Sylvester $\Delta(P_{\mathbf{A}}, Q_{\mathbf{A}})$ et $\Delta(P_{\mathbf{B}}, Q_{\mathbf{B}})$ ont les mêmes dimensions.

On étend l'homomorphisme F aux matrices à coefficients dans $\mathbf{A}$ et dans $\mathbf{B}$ en le notant toujours F. On a alors :

$$\begin{array}{rcl} F(\Delta(P_{\mathbf{A}}, Q_{\mathbf{A}})) & = & \Delta(P_{\mathbf{B}}, Q_{\mathbf{B}}) \\ F(|\Delta(P_{\mathbf{A}}, Q_{\mathbf{A}})|) & = & |\Delta(P_{\mathbf{B}}, Q_{\mathbf{B}})| \\ F(\mathrm{Res}(P_{\mathbf{A}}, Q_{\mathbf{A}})) & = & \mathrm{Res}(P_{\mathbf{B}}, Q_{\mathbf{B}}) \end{array}$$

On se place souvent dans le cas où $\mathbf{B}$ est un corps $\mathbf{K}$ et où $\mathbf{A}$ est l'anneau $\mathbf{K}[Y]$ des polynômes en une variable Y à coefficients sur $\mathbf{K}$. Ainsi $P_{\mathbf{K}[Y]}$ et $Q_{\mathbf{K}[Y]}$ sont dans $(\mathbf{K}[\mathbf{Y}])[X]$. Le résultant $\mathrm{Res}(P_{\mathbf{K}[Y]}, Q_{\mathbf{K}[Y]})$ sera donc dans $\mathbf{K}[Y]$.

L'homomorphisme F est l'évaluation en un point y donné de $\mathbf{K}$. On calculera le résultant (en X donc) dans $\mathbf{K}[Y]$ de $P_{\mathbf{K}[Y]}$ et $Q_{\mathbf{K}[Y]}$ pour obtenir un polynôme $R(Y)$.

Si y est un point de $\mathbf{K}$ qui annule $R(Y)$ alors le résultant (dans $\mathbf{K}$) de $P_{\mathbf{K}}$ et de $Q_{\mathbf{K}}$ est nul. Les polynômes $P_{\mathbf{K}} = P_{\mathbf{K}[Y]}(y)$ et $Q_{\mathbf{K}} = Q_{\mathbf{K}[Y]}(y)$ ont dans ce cas un pgcd non trivial dans $\mathbf{K}[X]$.

Une autre propriété des résultants est leur propriété multiplicative. Soient P_1, P_2 et Q des polynômes de $\mathbf{A}[X]$ on a

$$\mathrm{Res}(P_1P_2, Q) = \mathrm{Res}(P_1, Q)\mathrm{Res}(P_2, Q)$$

On peut en trouver une démonstration dans [Loo83a] ou [Lan69]. En général toutes les démonstrations concernant les résultants sont délicates car elles nécessitent de revenir à la définition en tant que déterminant de la matrice de Sylvester. Leur utilisation est par contre simple puisqu'il suffit de trouver « le bon résultant » c'est à dire celui dont les racines sont les objets cherchés.

1.3 Corps et extensions

Nous rappelons ici des définitions utiles pour modéliser les corps réels clos. Un corps réel clos $\mathbf{K}$ a toutes les propriétés « habituelles » de $\mathbb{R}$. On peut étendre l'analyse usuelle sur les fonctions de $\mathbb{R}^n$ dans $\mathbb{R}$ aux polynômes en n variables à coefficients dans $\mathbf{K}$.

Définition 4 (Corps réels et ordonnés)

1. Un corps $\mathbf{K}$ est dit *réel* si (-1) n'est pas une somme de carrés d'éléments de $\mathbf{K}$.
2. Un corps $\mathbf{K}$ est dit *ordonné* s'il possède une relation d'ordre total compatible avec l'addition et la multiplication par un élément positif
   - $\forall x, y, z \in \mathbf{K}\, x \leq y \Rightarrow x + z \leq y + z$
   - $\forall x, y, z \in \mathbf{K}\, x \leq y, z \geq 0 \Rightarrow xz \leq yz$

Un corps ordonné est toujours réel et un corps réel peut toujours être ordonné, il y a en général plusieurs ordres possibles dans un corps réel. On voit facilement que dans un corps ordonné un carré est toujours positif, par suite une nombre négatif ne peut avoir de recine carrée dans une corps ordonné.

Définition 5 (Extensions de corps)

1. Une *extension* $\mathbf{L}$ d'un corps $\mathbf{K}$ est un corps contenant $\mathbf{K}$. Une extension $\mathbf{L} \supset \mathbf{K}$ est dite *réelle* si $\mathbf{L}$ est un corps réel.
2. Une extension $\mathbf{L} \supset \mathbf{K}$ est dite *algébrique* si tous les éléments de $\mathbf{L}$ sont racine de polynômes de $\mathbf{K}[X]$. Elle est dite *algébrique réelle* si $\mathbf{L} \supset \mathbf{K}$ est réelle.

3. Une extension $\mathbf{L} \supset \mathbf{K}$ est dite *finie* si la dimension de $\mathbf{L}$ en tant qu'espace vectoriel sur $\mathbf{K}$ est finie. Le *degré* de l'extension $\mathbf{L} \supset \mathbf{K}$ est alors la dimension de $\mathbf{L}$ en tant que $\mathbf{K}$-espace vectoriel.

Si une extension est finie elle peut être vue comme l'anneau quotient

$$\mathbf{K}[X]/{<}P(X){>}$$

où $P(X)$ est un polynôme irréductible et où $<P(X)>$ désigne l'idéal des multiples de $P(X)$ dans $\mathbf{K}[X]$. Les éléments de $\mathbf{K}[X]/{<}P(X){>}$ sont les classes modulo $P(X)$ dont on prend classiquement un représentant réduit modulo $P(X)$.

L'arithmétique de $\mathbf{K}[X]/{<}P(X){>}$ se déduit facilement de l'arithmétique des polynômes réduits modulo $P(X)$. L'opération d'inversion correspond à la relation de Bezout. Soit Q une classe modulo $P(X)$ et $Q(X)$ un représentant réduit modulo $P(X)$ de Q. On calcule $\overline{Q(X)}$ et $\overline{P(X)}$ dans $\mathbf{K}[X]$ vérifiant

$$\begin{array}{rcl} Q(X)\overline{Q(X)} + P(X)\overline{P(X)} & = & 1 \quad \text{dans } \mathbf{K}[X] \\ Q\overline{Q} & = & 1 \quad \text{dans } \mathbf{K}[X]/{<}P(X){>} \end{array}$$

en posant $\overline{Q}$ la classe modulo $P(X)$ de $\overline{Q(X)}$. L'algorithme d'Euclide calcule directement $\overline{Q(X)}$ réduit modulo $P(X)$.

1.3.1 Éléments primitifs

Soient $\mathbf{K_1} \supset \mathbf{K}$ et $\mathbf{K_2} \supset \mathbf{K}$ deux extensions de corps engendrées respectivement par α_1, une racine du polynôme irréductible $P_1 \in \mathbf{K}[X]$, et α_2 une racine du polynôme irréductible $P_2 \in \mathbf{K}[X]$.

Le théorème de l'élément primitif (voir [Loo83a, Lan69]) permet de déterminer un *élément primitif* α_3 racine de $P_3 \in \mathbf{K}[X]$ générant $\mathbf{K_3} \supset \mathbf{K}$ contenant à la fois $\mathbf{K}_1$ et $\mathbf{K}_2$.

Décrivons en le fonctionnement. On calcule d'abord le résultant

$$P(X,T) \in \mathbf{K}[X,T] = \mathrm{Res}_Y(P_1(X-TY), P_2(Y))$$

Ce résultant exprime les valeurs de X et de T pour lesquelles $P_1(X - TY)$ et $P_2(Y)$ ont une racine commune. Cette racine vaut nécessairement α_2 puisqu'il faut annuler P_2. De même comme $P_1(X - T\alpha_2)$ doit s'annuler on doit avoir $X - T\alpha_2 = \alpha_1$ et l'élément primitif que l'on cherche aura la forme $X = \alpha_1 + T\alpha_2$.

On se place dans un corps algébriquement clos $\overline{\mathbf{K}}$ contenant $\mathbf{K}$. Soit un couple (x,t) dans $\overline{\mathbf{K}}$ annulant $P(X,T)$ (vu dans $\overline{\mathbf{K}}$) tel que t soit un entier (ce qui est toujours possible puisqu'un corps est toujours une $\mathbb{Z}$-algèbre et $t.1$ se note t). Dans ce cas $P_1(x-tY)$ et $P_2(Y)$ ont un facteur commun $G_{x,t}(Y)$ dans $\overline{\mathbf{K}}[Y]$. On veut exprimer α_1 et α_2 en fonction d'une racine de $G_{x,t}(Y)$.

On choisit t tel que $P(X,t)$ soit sans facteurs carrés en X. Tout entier t qui n'annule pas le discriminant

$$R(T) \in \mathbf{K}[T] = \mathrm{Res}_X(P(X,T), \frac{\partial P(X,T)}{\partial X}) \tag{1.8}$$

convient.

Soit $P_3(X)$ un facteur irréductible de $P(X,t)$ (dans $\mathbf{K}[X]$ puisque t est entier) et α_3 une racine de P_3 dans $\overline{\mathbf{K}}$. Plaçons nous dans l'extension $\mathbf{K}_3 \supset \mathbf{K}$ engendrée par α_3 la racine de P_3. On peut exprimer α_1 et α_2 en fonction de α_3 en calculant :

$$G_{\alpha_3,t}(X) \in \mathbf{K_3}[X] = \mathrm{pgcd}_X(P_1(\alpha_3 - tX), P_2(X))$$

Par construction α_2 est racine de $G_{\alpha_3,t}(X)$ puisqu'il annule $P_2(Y)$ et $P_1(\alpha_3 - tY))$. Écrivons $P_2(Y)$ sous la forme $(Y-\alpha_2)P_{\alpha_2}(Y)$ on alors

$$\begin{aligned} P(X,t) &= \mathrm{Res}_Y(P_1(X-tY), P_2(Y)) \\ &= \mathrm{Res}_Y(P_1(X-tY), (Y-\alpha_2)P_{\alpha_2}(Y)) \\ &= \mathrm{Res}_Y(P_1(X-tY), (Y-\alpha_2)) \\ &\quad\ \mathrm{Res}_Y(P_1(X-tY), P_{\alpha_2}(Y)) \end{aligned}$$

par multiplicativité du résultant. Si on se donne une autre racine α_2' de $G_{\alpha_3,t}(X)$ celle ci annule $P_{\alpha_2}(Y)$ et aussi $P_1(X-TY)$. On voit maintenant que α_3 est racine double du résultant $P(X,t)$ ce qui est impossible puisque t n'annule pas le discriminant 1.8.

On voit donc que α_2 est la seule racine de $G_{\alpha_3,t}$ dans $\overline{\mathbf{K}}$. Il a donc la forme $X - Q_1(\alpha_3)$ dans $(\mathbf{K}[\alpha_3])[X]$ en identifiant un élément de $\mathbf{K}_3 = \mathbf{K}/{<}P_3{>}$ à son représentant canonique dans $\mathbf{K_3}[X]$. Ce qui exprime $\alpha_1 = Q_1(\alpha_3)$ en fonction de α_3. On en déduit alors $\alpha_2 = \alpha_3 - tQ_1(\alpha_3)$.

On a ainsi exprimé α_1 et α_2 en fonction de α_3 à l'aide de calculs dans $\mathbf{K}[X]$[4], nous avons simplement « raisonné » dans $\overline{\mathbf{K}}[X]$. Il nous a suffit pour cela de calculer des résultants.

En pratique cette technique est limitée à des petits degrés puisque le polynôme de définition $P_3(X)$ est en général de degré nm si $P_1(X)$ est de degré n et $P_2(X)$ est de degré m. Par exemple si on a 10 nombres algébriques définis par des polynômes de degré 2 à gérer, un élément primitif α permettant

[4]ou $\mathbf{K}[X]/{<}P(X){>}$ ce qui revient au même

de les exprimer est défini par un polynôme de degré 1024. Ainsi l'exemple de Ramanujan

$$\sqrt[3]{-\sqrt[5]{\frac{27}{5}}+\sqrt[5]{\frac{32}{5}}} = \left(-\sqrt[5]{3}^2+\sqrt[5]{3}+1\right)\sqrt[5]{\frac{1}{25}}$$

donné dans [DST87] se décrit avec 4 racines cinquièmes[5] et une racine cubique. À notre connaissance aucune technique utilisant les éléments primitifs n'a jamais pu le résoudre parce qu'il faut factoriser un polynôme de degré 1875.

Nous verrons dans la section 2 d'autres méthodes qui s'apparentent aux éléments primitifs ainsi que les techniques que j'ai proposées, Ces dernières permettent de résoudre cet exemple en moins d'une minute de calcul.

1.3.2 Corps réels clos

Les corps réels clos sont par rapport aux polynômes en une variable l'analogue des nombres réels usuels par rapport aux suites de Cauchy.

Définition 6 (Corps réel clos)

1. Un corps $\mathbf{K}$ est dit *réel clos* s'il n'a pas d'extension algébrique non triviale qui soit réelle.
2. Un corps réel clos est toujours ordonné. Une définition équivalente d'un corps réel clos est un corps dans lequel
   - tout nombre positif a une racine carrée,
   - tout polynôme de degré impair admet au moins une racine.
3. Une autre définition équivalente est qu'un corps $\mathbf{K}$ est réel clos si l'extension de corps $\overline{\mathbf{K}} = \mathbf{K}[X]/ < X^2+1 >$ est algébriquement close.
4. Un corps ordonné $\mathbf{K}$ est toujours contenu dans un corps réel clos. On appelle *clôture réelle* de $\mathbf{K}$ et on note $\tilde{\mathbf{K}}$ le plus petit corps réel clos qui contient $\mathbf{K}$.

La clôture réelle $\tilde{\mathbf{K}}$ d'un corps ordonné $\mathbf{K}$ est « bien définie » (voir [BCR87]) dans le sens où elle est unique à un isomorphisme de corps ordonnés près. L'extension $\tilde{\mathbf{K}} \supset \mathbf{K}$ est algébrique mais elle n'est pas de dimension finie. Par contre tous les éléments de $\tilde{\mathbf{K}}$ sont algébriques et sont donc dans une extension de type fini de $\mathbf{K}$. Elle est « constructive » (voir [Hol41], [Lan69], [Loo83a]) dans le sens où on peut toujours exprimer des racines de polynômes de $\tilde{\mathbf{K}}[X]$ à l'aide de calculs dans $\mathbf{K}[X]$.

[5] en regardant de près 3 suffisent

L'analyse élémentaire des fonctions continues de la variable réelle se généralise aux fonctions polynomiale d'un corps réel clos. L'ordre sur le corps induit la topologie euclidienne habituelle de $\mathbb{R}$. Les éléments des réels clos ont toutes les propriétés des nombre réels usuels si on se restreint aux polynômes.

Par exemple le critère de changement de signe est vrai pour les polynômes. Si $\mathbf{K}$ est un corps réel clos et si un polynôme P de $\mathbf{K}[X]$ est négatif en un point a de $\mathbf{K}$ et positif en un point b de $\mathbf{K}$ (avec $a < b$), alors il existe un point x_0 de l'intervalle (a, b) dans $\mathbf{K}$ tel que P s'annule en x_0.

1.4 Algorithmes de base

Le théorème de Sturm et ses analogues permettent de compter ou d'isoler les racines d'un polynôme entre deux points. Différentes variantes permettent de calculer le signe d'une expression au dessus d'une de ses racines réelles. On ramène ainsi un problème algébrique dans $\tilde{\mathbf{K}}$ à plusieurs problèmes de calculs de suite de Sturm dans $\mathbf{K}[X]$ et à des calculs dans $\mathbf{K}$. Ainsi calculer le signe d'une expression $Q(X) \in \mathbf{K}$ « au dessus » d'un point $\alpha \in \tilde{\mathbf{K}}$ revient à évaluer le signe de $Q(\alpha)$ vu dans $\tilde{\mathbf{K}}[X]$ puis évalué dans $\tilde{\mathbf{K}}$ en ne faisant des calculs que dans $\mathbf{K}$ (ou $\mathbf{K}[X]$ ce qui revient au même). On peut ainsi « simuler » des calculs dans $\tilde{\mathbf{K}}$.

1.4.1 Suite de Sturm

Définition 7 (Suite de Sylvester)
Soit $\mathbf{K}$ un corps, soit P et Q deux polynômes de $\mathbf{K}[X]$ avec $P \neq 0$. La *suite de Sylvester* $\mathcal{S}_{P,Q} = S_0, S_1, \ldots S_k$ des polynômes P et Q est définie par

$S_0 = P$
$S_1 = Q$
$S_{i-1} = Q_i S_i - S_{i+1}$ pour $0 \leq i \leq k$
où Q_i est le quotient de la division euclidienne de S_{i-1} par S_i
$S_{k+1} = 0$

C'est aux signes près la suite calculée par l'algorithme d'Euclide puisque S_{i+1} est l'opposé du reste de la division euclidienne de S_{i-1} par S_i.

La *suite de Sturm* $\mathcal{S}_{P,P'}$ d'un polynôme P est par définition la suite de Sylvester de P et de sa dérivée P'.

L'intérêt de la suite de Sylvester $S_0, S_1, \ldots S_k$ de deux polynômes est de conserver les signes lorsqu'on traverse une racine x_0 de S_i. En effet si x_0 est une racine de S_i on a $S_{i-1}(x_0) = -S_{i+1}(x_0)$ pour $0 < i \leq k-1$ et les polynômes S_{i-1} et S_{i+1} sont de signe contraire en x_0. Par continuité cette propriété se conserve dans un voisinage de x_0.

FIG. 1.1 – Variations de $\mathcal{S}_{P,Q}$ en S_i

	x_{i-}	x_i	x_{i+}
$S_0 \dots S_{i-1}$	$\mathcal{V}_{i,l}$	$\mathcal{V}_{i,l}$	$\mathcal{V}_{i,l}$
S_{i-1}	s_{i-1}	s_{i-1}	s_{i-1}
S_i	s_{i-}	0	s_{i+}
S_{i+1}	s_{i+1}	s_{i+1}	s_{i+1}
$S_{i+1} \dots S_n$	$\mathcal{V}_{i,u}$	$\mathcal{V}_{i,u}$	$\mathcal{V}_{i,u}$
$\mathcal{S}_{P,Q}$	$\mathcal{V}_{i-}$	$\mathcal{V}_i$	$\mathcal{V}_{i+}$

Définition 8 (Variation de signes)
Soit $(f_i)_{i=0}^{i=k} = f_0, \dots, f_i, \dots f_k$ une suite de nombres avec f_0 non nul. On construit la suite $(g)_{i=0}^{i=l}$ à partir de f en enlevant les termes nuls de f. Si $g_0 = f_0$ et $g_i = f_{j_i}$, $g_{i+1} = f_{j_i+j}$ où j est le plus petit indice tel que $f_{j_i+j} \neq 0$.

La *variation de signes* de f est le nombre de changements de signes que comporte la suite g. On compte un changement de signe à chaque fois que $\mathtt{sgn}(g_i) \neq \mathtt{sgn}(g_{i+1})$.

Nous ne définissons pas pour l'instant la variation de signe d'une suite dont le premier élément est nul.

Soit donc $\mathcal{S}_{P,Q}$ la suite de Sylvester de deux polynômes P et Q. Nous allons nous intéresser à la variation de signe de ses évaluations en différents points de la droite réelle. Celle variation ne peut changer qu'en traversant une racine d'un des polynômes S_i de $\mathcal{S}_{P,Q}$.

Pour simplifier nous pouvons supposer que P et Q sont premiers entre eux. On pourra toujours se ramener à ce cas. Deux polynômes consécutifs S_i et S_{i+1} de $\mathcal{S}_{P,Q}$ ne peuvent donc avoir de racine commune. En effet si on annule S_i et S_{i+1} en un point x on annule aussi S_{i-1} en ce point et par suite P et Q ont x comme racine commune.

Faisons le tableau 1.1 de la variation de signe de $\mathcal{S}_{P,Q} = S_0, S_1 \dots S_n$. Plaçons nous au voisinage d'une racine x_i de S_i. Appelons x_{i-} et x_{i+} des points respectivement à gauche et à droite de x_i qui soient suffisamment proches de x_i pour qu'aucun autre polynôme de $\mathcal{S}_{P,Q}$ ne s'annule dans l'intervalle.

Dans le tableau de variations de la figure 1.1 on voit qu'en x_i, les signes s_{i-1} et s_{i+1} ne peuvent être nuls puisque P et Q sont premiers entre eux. On voit donc que la variation de signe $\mathcal{V}_i$ de $\mathcal{S}_{P,Q}$ en x_i vaut $\mathcal{V}_{i,l} + \mathcal{V}_{i,u}$ dans le cas où s_{i-1} vaut S_i et $\mathcal{V}_{i,l} + \mathcal{V}_{i,u} + 1$ dans le cas contraire.

On voit maintenant que quels que soient les signes non nuls s_{i-} et s_{i+} de S_i en x_i, la variation de signe $\mathcal{V}_{i-}$ de $S_0 \dots S_{i-1}$ en x_{i-} vaut $\mathcal{V}_i$. De même la

FIG. 1.2 – tableau de signes de $\mathcal{S}_{P,P'}$

	x_-	x	x_+
P	s_-	0	s_+
P'	s'	s'	s'
$S_1 \ldots S_n$	$\mathcal{V}'$	$\mathcal{V}'$	$\mathcal{V}'$
$\mathcal{S}_{P,P'}$	$\mathcal{V}_-$	$\mathcal{V}$	$\mathcal{V}_+$

variation de signe $\mathcal{V}_{i+}$ de $S_{i+1} \ldots S_n$ en x_{i-} vaut $\mathcal{V}_i$. Ce que l'on peut résumer en disant que la variation de signe de $\mathcal{S}_{P,Q}$ ne change pas quand on traverse une racine x_i de S_i avec $i > 0$.

Pour étudier la variation de signe de la suite de Sylvester $\mathcal{S}_{P,Q}$ de deux polynômes P et Q premiers entre eux il suffit donc d'examiner localement le signe du premier polynôme P au voisinage d'une de ses racines.

1.4.2 Comptage des racines

On peut maintenant s'intéresser à la suite de Sturm $\mathcal{S}_{P,P'}$ de P et examiner localement les changements de signes de la suite en une racine x de P. La suite $\mathcal{S}_{P,P'}$ s'écrit $P, S_1 \ldots S_n$ où $S_1 \ldots S_n$ commence par P'. On peut dresser le tableau de variations 1.2 qui permet de compter les racines de P entre deux points a et b.

On voit clairement que si le signe s_- de P en x_- est négatif alors le signe s' de P' en x est positif ainsi que s_+ le signe de P en x. La variation $\mathcal{V}_-$ de $\mathcal{S}_{P,P'}$ en x_- vaut donc $1 + \mathcal{V}_+$. De la même façon si s_- est positif alors s' et s_+ sont positifs et on a toujours $\mathcal{V}_- = 1 + \mathcal{V}_+$.

Nous avons donc montré que la variation de signe augmente de 1 lorsque l'on traverse une racine de P dans le sens croissant. Nous en déduisons immédiatement

Proposition 1 (Théorème de Sturm)
Soient $\mathbf{K}$ un corps réel clos, $P(X)$ un polynôme sans facteurs carrés de $\mathbf{K}[X]$, x_1 et x_2 deux points de $\mathbf{K}$ avec $x_1 < x_2$ qui ne sont pas racines de P et $\mathcal{S}_{P,P'}$ la suite de Sturm de P.

Le nombre de racines de P dans l'intervalle $]x_1, x_2[$ de $\mathbf{K}$ est $v_2 - v_1$ où v_1 (resp v_2) est le nombre de changements de signes de la suite S_1 (resp S_2) obtenue en évaluant $\mathcal{S}_{P,P'}$ en x_1 (resp x_2).

En général on se ramène au cas où P est sans facteurs multiples en calculant le pgcd de P et de P' lorsque P n'est pas sans facteurs carrés.

Nous pouvons maintenant étendre la définition 8 pour prendre en compte le signe de P en x.

Définition 9
Soit $(f_i)_{i=0}^{i=k} = f_0, \ldots, f_i, \ldots f_k$ une suite de nombres non tous nuls. On considère la suite $(g)_{i=0}^{i=l}$ sous suite de f qui commence à l'indice j qui est le plus petit indice tel que f_j soit non nul. Soit v la variation de signe de $(g)_{i=0}^{i=l}$ au sens de la définition 8. La *variation de signe* de f vaut v si j vaut 0 et $v+1$ dans le cas contraire.

Dans le cas où $(f_i)_{i=0}^{i=k}$ est issue de l'évaluation d'une suite de deux polynômes de degrés positifs premiers entre eux on peut facilement voir que $k \geq 2$ et que $j \leq 1$. La suite $(g)_{i=0}^{i=l}$ contient donc toujours deux termes.

Nous pouvons généraliser le théorème de Sturm au cas où x_1 est une racine de P. Il suffit pour cela d'examiner le tableau 1.2 pour s'apercevoir que $\mathcal{V}$ vaut $\mathcal{V}_-$ en x avec la nouvelle définition.

Proposition 2
Soient $\mathbf{K}$ un corps réel clos, $P(X)$ un polynôme sans facteurs carrés de $\mathbf{K}[X]$, x_1 et x_2 deux points de $\mathbf{K}$ avec $x_1 < x_2$ et $\mathcal{S}_{P,P'}$ la suite de Sturm de P.

Le nombre de racines de P dans l'intervalle $[x_1, x_2[$ de $\mathbf{K}$ est $v_2 - v_1$ en reprenant les notations de 1 avec la définition 9 pour la variation de signe.

Cette généralisation est celle que j'avais proposée dans ma thèse (voir [Rio91]).

1.4.3 Calcul de signes

Pour un polynôme $P \in \mathbf{K}[X]$ n'ayant qu'une racine réelle $x \in \tilde{\mathbf{K}}$ entre deux points $a \in \mathbf{K}$ et $b \in \mathbf{K}$ d'un corps ordonné $\mathbf{K}$ contenu dans un corps réel clos $\tilde{\mathbf{K}}$ avec $a \leq x < b$ dans $\tilde{\mathbf{K}}$ nous pouvons examiner le signe d'une expression $Q(x) \in \tilde{\mathbf{K}}$ en faisant simplement des calculs dans $\mathbf{K}[X]$.

La méthode proposée dans [BPR96] est de calculer la suite de Sylvester de P et de $P'Q$. Cette technique fait inutilement grossir les calculs quand on caractérise x par un intervalle d'isolation $[a, b[$, ne contenant pas d'autre racine de P que x. En effet, le signe de P' est connu sur l'ensemble de l'intervalle et il est plus simple de faire le tableau de variations 1.3. Le signe de $Q(x_0) \in \tilde{\mathbf{K}}$ s'en déduit facilement.

On peut alors garantir que si on calcule le signe d'une expression simple (dont le degré est petit par exemple), le calcul de la suite de Sylvester de P et de Q restera simple. Ce qui ne serait pas le cas si on calculait la suite de Sylvester de $P'Q$ et de P puisqu'en général $P'Q \bmod (P)$ est de degré

FIG. 1.3 – Variations de $\mathcal{S}_{P,Q}$ en P

(a)	x_-	x	x_+
P	$-$	0	$+$
Q	$+$	$+$	$+$
$\mathcal{S}_{P,Q}$	v	v	$v-1$

(b)	x_-	x	x_+
P	$-$	0	$+$
Q	$-$	$-$	$-$
$\mathcal{S}_{P,Q}$	v	$v+1$	$v+1$

(c)	x_-	x	x_+
P	$+$	0	$-$
Q	$+$	$+$	$+$
$\mathcal{S}_{P,Q}$	v	$v+1$	$v+1$

(d)	x_-	x	x_+
P	$+$	0	$-$
Q	$-$	$-$	$-$
$\mathcal{S}_{P,Q}$	v	v	$v-1$

maximal et la suite contient plus de termes. Comme il faudra ensuite évaluer les polynômes et calculer les signes de ces évaluations, on devra en évaluer plus pour la suite $\mathcal{S}_{P,P'Q}$ que pour la suite $\mathcal{S}_{P,Q}$. Nous verrons dans la section 3.4 comment généraliser ce résultat pour un anneau.

Chapitre 2

La clôture réelle

D'un point de vue effectif, étant donné un corps ordonné $\mathbf{K}$ nous devons pouvoir calculer dans le corps réel clos $\tilde{\mathbf{K}} \supset \mathbf{K}$. Pour cela nous devons autoriser l'utilisateur à ajouter des racines de polynômes à $\tilde{\mathbf{K}}$.

Voyons comment on manipulait les nombres algébriques réels avant la clôture réelle. Un nombre algébrique α est représenté par un couple (I_α, P_α) où P_α est un polynôme de $\mathbb{Q}[X]$ et où I_α est un intervalle d'isolation dont les bornes sont rationnelles. Il a donc la forme $I_\alpha = (l_\alpha, r_\alpha)$ où l_α et r_α sont dans $\mathbb{Q}$. On choisit P_α sans facteurs carrés et I_α (vu comme intervalle de $\mathbb{R}$) ne contenant qu'une seule racine de P_α. L'arithmétique sur ces représentations utilise des techniques d'éléments primitifs qui sont de calculer un résultant à chaque fois que l'on fait une opération

Voyons par exemple comment implanter l'addition sur cette représentation. On doit représenter $\alpha = \alpha_1 + \alpha_2$ où α_1 est représenté par I_1, P_1 et α_2 par I_2, P_2. On cherche I_α, P_α représentant α.

La méthode décrite dans [Loo83a] et qui est reprise dans MATHEMATICA (voir [Str97]) est de calculer le résultant

$$R_\alpha = \mathrm{Res}_Y(P_1(X - Y), P_2(Y))$$

qui est un polynôme de $\mathbb{Q}[X]$. On voit que α est racine de R_α, en effet si on évalue dans $\mathbb{R}[X]$ le polynôme $(P_1(X - Y)$ en $X = \alpha = \alpha_1 + \alpha_2$ on trouve $(P_1(\alpha_1 + \alpha_2 - Y)$ qui a α_2 comme racine. Le polynôme $P_2(Y)$ a aussi α_2 comme racine et par suite $(P_1(\alpha_1 + \alpha_2 - Y)$ et $P_2(Y)$ on un pgcd non trivial dans $\mathbb{Q}[X]$. Ceci prouve que le résultant R_α s'annule en α.

On peut maintenant choisir pour P_α la partie sans facteurs carrés de R_α. Pour trouver un intervalle d'isolation de α on regarde si l'intervalle $I_\alpha = (l_1 + l_2, r_1 + r_2)$ ne contient qu'une racine. Si ce n'est pas le cas on peut resserrer chacun des intervalles I_1 et I_2 jusqu'à ce que $(l_1 + l_2, r_1 + r_2)$ ne contienne plus qu'une racine de P_α. On alors un intervalle I_α d'isolation pour α.

Les autres opérations se font de manière analogue (voir [Loo83a]) et beaucoup d'heuristiques peuvent être implantées pour améliorer les temps de calculs (voir [Str97]). D'un point de vue pratique ces techniques on l'inconvénient de ne pas pouvoir faire facilement des simplifications. Prenons par exemple le nombre $1+\sqrt{3}$ dont le polynôme de définition est X^2-2X-2 et soustrayons lui le nombre $\sqrt{3}$ dont le polynôme de définition est X^2-3. Nous sommes amenés à considérer le polynôme

$$X^4-4X^3-6X^2+20X-11=(X-1)^2(X^2-2X-11)$$

Pour nous apercevoir que le résultat vaut 1, il faut au moins faire la factorisation sans carrés complète.

2.1 Les tours d'extensions

Plutôt que d'utiliser des techniques d'éléments primitifs nous pouvons travailler dans une suite d'extensions.

$$\tilde{\mathbf{K}}=\mathbf{K}_\infty\cdots\supset\mathbf{K_n}\supset\ldots\mathbf{K}=\mathbf{K_0}$$

On évite ainsi les factorisation de polynômes qui peuvent rapidement devenir gros. Nous avons proposé avec Marie Françoise Roy et Zenon Ligatsikas (voir [LRR96]) un modèle à base de tours d'extensions. La méthode s'inspirait de celle de l'évaluation dynamique de Dominique Duval (voir [DDDD85]) en évitant la discussion de cas. Dans [DDDD85], lorsqu'un polynôme $P_\alpha(X)$ définissant un nombre algébrique α est réductible et que l'on veut tester à zéro où inverser une expression $Q(\alpha)$, on calcule le pgcd $G(X)$ de $P_\alpha(X)$ et de $Q(X)$. On a maintenant deux cas possibles :

- le cas où $G(\alpha)$ s'annule et
- le cas où $\frac{P}{G}(\alpha)$ s'annule.

Dans le premier cas on peut dire que $Q(\alpha)$ est nul (non inversible). Dans le second cas on peut dire que le nombre algébrique α est défini par $P(X)/G(X)$[1] ce qui simplifie le polynôme de définition de α. Ainsi dans le cas où $P(X)$ est sans facteurs carrés on sait que $\frac{P}{G}(X)$ et $Q(X)$ sont premier entre eux et on peut répondre que $Q(\alpha)$ n'est pas nul ou calculer explicitement son inverse.

Dans le cas réel la situation est « plus simple » puisqu'un seul des deux cas est possible. En effet une même valeur réelle ne peut être racine de deux

[1] qui est bien un polynôme

polynômes premiers entre eux. Si l'on dispose d'un moyen effectif pour désambiguïser les deux cas, les calculs peuvent se continuer sans avoir à gérer une « discussion de cas ». C'est ce principe qui nous a guidé pour proposer dans [LRR96] un schémas où la partie algorithmique qui permet de désambiguïser les cas est séparée de la partie construction de tours d'extensions.

Dans [LRR96] nous avons pu proposer un modèle uniforme qui intègre dans un même programme les méthodes de caractérisation des racines réelles basées sur des intervalles et celles plus génériques basées sur le lemme de Thom [CR88, GVRRT98]. Nous avons ainsi pu vérifier expérimentalement que, malgré leur bonne complexité théorique, les méthodes basées sur le lemme de Thom nécessitaient trop de ressources et étaient plus lentes en pratique.

Ce modèle est aujourd'hui intégré à la distribution d'AXIOM et a l'avantage de pouvoir intégrer rapidement de nouveaux algorithmes ou des améliorations. Nous décrivons ici une adaptation du principe de fonctionnement que nous avions donné avec Marie Françoise Roy et Zenon Ligatsikas dans [LRR96]. J'avais en effet écrit le programme AXIOM pour pouvoir comparer différents algorithmes et différentes heuristiques de calcul.

L'adaptation décrite ici reflète ce que j'ai décrit dans [Rio02] pour mieux prendre en compte les simplifications.

2.2 Un modèle abstrait

L'idée générale de [LRR96] pour manipuler les nombres algébriques réels est assez simple. Cela revient à « abstraire » les opérations qui doivent être utilisées pour mener à bien des calculs dans les extensions algébriques. Ici $\mathbf{K}$ désigne une implémentation quelconque d'un corps ordonné et $\mathbf{K}[X]$ désigne une implémentation quelconque de polynômes en une variable à coefficients sur $\mathbf{K}$. Nous noterons $\{X \leftarrow \alpha\}_{\mathbf{K}}$ une implantation qui code des racines réelles de tous les polynômes de $\mathbf{K}[X]$.

Une racine réelle est un élément de $\{X \leftarrow \alpha\}_{\mathbf{K}}$, on la notera $X \leftarrow \alpha$ ou simplement α et permet de représenter un nombre algébrique réel de $\tilde{\mathbf{K}}$ (en travaillant dans $\mathbf{K}$ bien sûr). Le sous corps $\mathbf{K}_\alpha$ de $\tilde{\mathbf{K}}$ engendré par α pourra alors être noté comme l'anneau de polynômes $\mathbf{K}[X \leftarrow \alpha]$. On identifiera souvent $X \leftarrow \alpha$ dans $\{X \leftarrow \alpha\}_{\mathbf{K}}$ et α dans $\tilde{\mathbf{K}}$.

Nous utiliserons des opérations dont les signatures portent sur des « racines réelles ». Celles ci sont spécifiées dans une liste de signatures[2] nommée `real_root_characterisation`. Les signatures d'opérations dont nous avons besoin pour calculer avec des racines réelles sont :

[2]une catégorie AXIOM ou une espèce FoC.

– construire des racines à partir d'un polynôme de $\mathbf{K}[X]$:

$$\texttt{all\_roots\_of} \in \mathbf{K}[X] \to \texttt{list}(\{X \leftarrow \alpha\}_{\mathbf{K}})$$

L'appel $\texttt{all\_roots\_of}(P)$ retourne les racines réelles

$$X \leftarrow \alpha_1, \ldots X \leftarrow \alpha_n$$

représentant les nombres réels $\alpha_1, \ldots \alpha_n$ qui sont racines de P.

– Retourner un polynôme de définition pour une racine réelle :

$$\texttt{defining\_polynomial} \in \{X \leftarrow \alpha\}_{\mathbf{K}} \to \mathbf{K}[X]$$

L'appel $\texttt{defining\_polynomial}(\alpha)$ retourne le polynôme P_α de définition de α. Notons que le polynôme sera simplement supposé sans facteurs carrés.

– Réduire une expression de $\mathbf{K}[X]$ modulo un polynôme de définition pour une racine réelle :

$$\texttt{reduce} \in \mathbf{K}[X] \to \{X \leftarrow \alpha\}_{\mathbf{K}} \to \mathbf{K}[X]$$

L'appel $\texttt{reduce}(Q, \alpha)$ peut être implémenté par défaut en calculant le reste de la division euclidienne de Q par le polynôme de définition P_α de la racine réelle α.

– Décider si une expression de $\mathbf{K}[X]$ est nulle en un point réel α représenté par $X \leftarrow \alpha$:

$$\texttt{is\_zero} \in \mathbf{K}[X] \to \{X \leftarrow \alpha\}_{\mathbf{K}} \to (\texttt{bool} \oplus \{X \leftarrow \alpha\}_{\mathbf{K}})$$

L'appel $\texttt{is\_zero}(Q(X), X \leftarrow \alpha)$ retourne en général un booléen qui est vrai si l'expression $Q(X)$ s'annule au point α représenté par $X \leftarrow \alpha$. Comme nous le verrons dans la section 2.2.3, ces calculs sont longs et nécessitent un calcul de pgcd dans $\mathbf{K}[X]$. On peut trouver un diviseur du polynôme P_α annulé par α pendant le calcul. Dans ce cas on choisit de retourner un codage $X \leftarrow \alpha'$ du même nombre réel α dont le polynôme de définition P'_α est de degré plus petit que P_α. On réduit ainsi dynamiquement les degrés des polynômes qui interviennent.

– calculer le signe d'une expression de $\mathbf{K}[X]$ en point réel α représenté par $X \leftarrow \alpha$:

$$\texttt{sign} \in \mathbf{K}[X] \to \{X \leftarrow \alpha\}_{\mathbf{K}} \to (\texttt{int} \oplus \{X \leftarrow \alpha\}_{\mathbf{K}})$$

L'appel $\texttt{sign}(Q(X), X \leftarrow \alpha)$ retourne le signe (-1 ou 1) de l'expression $Q(X)$ au point réel α que représente $X \leftarrow \alpha$. C'est donc le signe de

$Q(\alpha)$ dans $\tilde{\mathbf{K}}$. De même que pour la fonction `is_zero` on choisira de retourner un meilleur codage lorsqu'on trouve un diviseur du polynôme de définition P_α de α. Ceci est en particulier le cas si $Q(\alpha)$ s'annule dans $\tilde{\mathbf{K}}$.

– Donner l'inverse d'une expression de $\mathbf{K}[X]$:

$$\texttt{recip} \in \mathbf{K}[X] \to \{X \leftarrow \alpha\}_{\mathbf{K}} \to (\mathbf{K}[X] \oplus \{X \leftarrow \alpha\}_{\mathbf{K}})$$

Si $X \leftarrow \alpha$ représente le nombre réel α, l'appel $\texttt{recip}(Q(X), X \leftarrow \alpha)$ retourne l'inverse s'il existe de $Q(\alpha)$. On retourne un meilleur codage quand on rencontre un diviseur d'un polynôme de définition.

2.2.1 Opérations Génériques

À titre d'exemple voyons comment nous pourrions implémenter l'opération $\texttt{recip}(Q, \alpha)$. Nous noterons P_α le polynôme de définition de α, `bezout` l'opération qui calcule les coefficients de Bezout de deux polynômes. Pour deux polynômes P et Q elle retourne un triplet $(\overline{P}, \overline{Q}, G)$ de polynômes de $\mathbf{K}[X]$ tels que

$$\overline{P}P + \overline{Q}Q = G$$

où G est le pgcd de P et de Q. Nous supposerons que le polynôme Q que l'on veut inverser au dessus de la racine α est réduit modulo P_α.

- Soit $(\overline{P_\alpha}, \overline{Q}, G) = \texttt{bezout}(P_\alpha, Q)$
- Si G est trivial on peut retourner le résultat $\frac{\overline{Q}}{\texttt{lc}(G)}$. Notons que l'on ne suppose pas que le pgcd est unitaire.
- Si G est non trivial on peut tester si G est nul au dessus de α en appelant l'opération `is_zero` de $\{X \leftarrow \alpha\}_{\mathbf{K}}$.
- L'appel $\texttt{is\_zero}(G, \alpha)$ doit retourner un nouveau codage α' du même nombre algébrique. On peut ainsi retourner α'.

  Nous laissons à l'application qui appelle la fonction `recip` le soin de réduire éventuellement le polynôme Q passé à `recip`. Comme P_α est supposé sans facteurs carrés, le polynôme de définition $P_{\alpha'}$ qui en est un facteur l'est aussi. Il vaut donc soit le polynôme G calculé par `bezout`, soit $\frac{P_\alpha}{G}$. Si l'appel $\texttt{recip}(Q, \alpha)$ retourne un nouveau codage α' nous sommes ainsi sûrs que $P_{\alpha'}$ est soit un diviseur de Q soit premier avec Q. Dans le premier cas Q se réduit à 0 modulo $P_{\alpha'}$, dans le deuxième cas un autre appel à `recip` retournera un résultat.

2.2.2 Un modèle en FoC

Nous utilisons les conventions de notation de la section 5.2 qui correspondent à un usage plus proche d'OCAML que d'AXIOM[3]. En particulier `self` désigne une instance quelconque d'un codage de racines réelles. Nous pouvons écrire les spécifications FoC :

```
type somme('a,'b) =
  | Res in 'a -> somme('a,'b)
  | New in 'b -> somme('a,'b) ;;

species racine_reelle
   (k is corps_ordonné,
    k_x is polynomes_formels_univaries(k))

   inherits basic_object =

   sig defining_polynomial in self -> k_x ;
   sig reduce in k_x -> self -> k_x ;
   sig all_roots_of in k_x -> List(self) ;

   sig is_zero in k_x -> self -> somme(bool,self) ;
   sig sign in k_x -> self -> somme(int,self) ;
   sig recip in k_x -> self -> somme(k_x,self) ;

   end
```

2.2.3 Le codage par intervalles

La version actuelle de la clôture réelle représente une racine réelle α par un couple formé d'un intervalle d'isolation I_α et un polynôme de définition P_α pour α. Les algorithmes utilisés sont ceux que j'ai décrits dans [Rio91]. Elle est distribuée avec les versions d'AXIOM et le code est disponible (voir [Rio])

Pour déterminer si une expression $Q(X)$ s'annule en α nous calculons le pgcd $G(X)$ de $Q(X)$ et de $P_\alpha(X)$.

- Si celui ci est trivial $Q(\alpha)$ n'est pas nul.
- Si G est non trivial on regarde s'il s'annule dans l'intervalle I_α. Il suffit pour cela de regarder s'il change de signe entre les bornes de l'intervalle.

[3] nous n'avons pas voulu introduire AXIOM

En effet, G est sans facteur carrés comme diviseur de P_α qui est sans facteurs carrés.

Pour construire un ensemble de racines réelles à partir d'un polynôme P on commence par mettre P sans facteurs carrés. Soit P_α ce polynôme, on détermine ensuite une borne pour les racines de P_α ce qui nous permet d'avoir un intervalle I contenant toutes les racines de P_α. On calcule alors la suite de Sturm de P_α et on compte le nombre de racines de P_α dans I grâce à la proposition 1. Si I contient plus d'une racine on coupe I en deux sous intervalles I_1 et I_2 et on cherche dans chacun de ces intervalles. Comme P_α est sans facteurs carrés on arrive toujours à trouver un intervalle ne contenant qu'une seule racine.

Il nous faut ensuite calculer le signe d'une expression $Q(X)$ en α. La technique que nous utilisons dans la version Axiom actuelle de la clôture réelle repose sur la règle de Descartes et sur des changements de variables.

Proposition 3 (Règle de Descartes)
Le nombre de racines strictement positives d'un polynôme diffère par un entier positif pair du nombre de changement de signe des coefficients du polynôme.

En particulier si tous les coefficients d'un polynôme P sont de même signe alors P n'a pas de racines strictement positives.

Pour $a < b$, le changement de variable

$$X \to \frac{bX + a}{X + 1}$$

permet de transformer l'intervalle $[a, b[$ en l'intervalle $[0, +\infty[$. Les racines de Q entre a et b sont alors transformées en les racines de

$$Q^*_{a,b}(X) = (X + 1)^n Q(\frac{bX + a}{X + 1})$$

entre 0 et $+\infty$.

Si $Q(\alpha)$ est non nul, on peut toujours trouver un intervalle $I_\alpha = [a, b[$ d'isolation autour de α tel que $Q^*_{a,b}$ n'ait plus de racines positives. Il nous faut donc d'abord vérifier que $Q(\alpha)$ est non nul puis examiner si $Q^*_{a,b}$ vérifie le critère de Descartes. Si ce n'est pas le cas on doit resserrer (par dichotomie par exemple) l'intervalle autour de α et recommencer en calculant $Q^*_{a,b}$ pour le nouvel intervalle $I_\alpha = [a, b[$.

En pratique le nombre de bissections que l'on doit faire dépend de $Q(\alpha)$ et s'il est proche de 0 on doit en faire beaucoup. Il faut donc calculer beaucoup de signes puisqu'on doit calculer $Q^*_{a,b}$ et les signes de ces coefficients à chaque étape.

2.3 Extensions de corps ordonnés

2.3.1 Les extensions simples

Une fois les opérations de racine réelle spécifiées nous pouvons construire une extension de corps $\mathbf{K}_\alpha \supset \mathbf{K}$ qui contient une racine réelle α. Nous avons besoin de préciser :

- un corps $\mathbf{K}$ noté « `k` » dans le code,
- une implémentation $\mathbf{K}[X]$ de polynômes en une variable à coefficients sur $\mathbf{K}$. Il est noté « `k_x` » dans le code,
- une implémentation d'un domaine de codage $\{X \leftarrow \alpha\}_\mathbf{K}$ de racines réelles. On le notera « `r_cod` » dans le code,
- une valeur α de $\{X \leftarrow \alpha\}_\mathbf{K}$ notée `racine` dans le code.

Nous pouvons maintenant implémenter les opérations nécessaires en appelant simplement l'opération correspondante du domaine de codage. En utilisant la syntaxe de FoC (où `!` désigne l'appel de méthode), nous avons :

```
species extension_ordonnée
   (k is corps,
    k_x is polynomes_formels_univaries(k),
    r_cod is racine_reelle(k,k_x),
    racine in r_cod)
```

Ici nous choisissons de représenter les expressions algébriques comme des polynômes en une variable à coefficients sur les corps de base. Nous voulons bien sûr implémenter un corps ordonné.

```
 inherits corps_ordonne =

   rep = k_x;
```

Afin de pouvoir tirer parti dynamiquement des simplifications nous avons besoin d'un « état interne » pour disposer en permanence du « meilleur codage ». Nous choisissons de manipuler cet état dans une référence mutable nommée `mutable`. Les fonctions `#ref`, `#deref` et `#set` sont directement importées d'OCAML et permettent respectivement

- de créer une référence,
- d'obtenir le contenu d'une référence,
- de modifier le contenu d'une référence.

Nous ne voulons pas que ces opérations soient visibles par d'autres programmes, elles sont donc locales.

```
local let mutable = #ref(racine) ;
local let la_racine = #deref(mutable);
local let set_racine(new_racine) =
  #set(!mutable , new_racine) ;
```

Nous pouvons alors implémenter l'addition par exemple :

```
let plus(n1,n2) =
  let r = k_x!plus(n1,n2) in
  let is_new_racine = r_cod!is_zero(r,!la_racine) in
  match is_new_racine with
    | #Bool(test) ->
      if test then !zero else r_cod!reduce(r,!la_racine)
    | #New(new_racine) ->
      let dumb = !set_racine(new_racine) in
        r_cod!reduce(r,!la_racine) ;
```

Ici nous examinons « par filtrage » le résultat de l'appel de l'opération `is_zero` qui est soit un booléen, soit un nouveau codage.

- Lorsque le résultat est un booléen il suffit de simplifier « `r` ». Ceci est nécessaire puisque entre le calcul de « `n1` » ou « `n2`» et l'addition que nous effectuons un effet de bord sur `!la_racine` a pu se produire.
- Lorsque le résultat est un nouveau codage `new_racine`, cela traduit le fait que « `r` » n'était pas premier avec le polynôme de définition de `!la_racine` et nous devons faire une mise à jour avant de réduire modulo le nouveau polynôme de définition.

Les autres opérations se font de manière analogue. En particulier, si α est la racine réelle qui engendre $\mathbf{K}_\alpha$, le calcul du signe d'une expression $Q(\alpha)$ représentée par le polynôme $Q(X)$ appelle l'opération `sign`$(Q(X), \alpha)$ de $\{X \leftarrow \alpha\}_\mathbf{K}$, celui de son inverse appelle `recip`$(Q(X), \alpha)$.

Nous pouvons noter que ce sont les opérations arithmétiques qui appellent l'opération `is_zero` de $\{X \leftarrow \alpha\}_\mathbf{K}$. Le test d'une expression de $\mathbf{K}_\alpha$ à 0 est alors trivial. Lorsque une opération arithmétique retourne un polynôme Q non constant on peut supposer que celui ci est premier avec le polynôme de définition de α.

D'un point de vue pratique cette technique s'est avérée plus rapide en AXIOM que celle qui consisterait à appeler `is_zero` à chaque fois que l'on veut tester une expression à 0.

2.3.2 Les extensions multiples

Pour décrire des extensions contenant plusieurs racines réelles il suffit d'itérer la construction précédente et donc de construire l'extension $\mathbf{K}_{\alpha,\beta} \supset \mathbf{K}_\alpha \supset \mathbf{K}$. Par exemple soit α la plus grande racine réelle de $X^2 - X - 1$. Soit $\sqrt{5}$ la plus grande racine réelle de $X^2 - 5$. On souhaite vérifier que

$$2\alpha = \sqrt{5} + 1 \tag{2.1}$$

On peut soit se placer dans l'extension $\mathbf{K}_{\alpha,\sqrt{5}} \supset \mathbf{K}_\alpha \supset \mathbf{K}$ soit dans l'extension $\mathbf{K}_{\sqrt{5},\alpha} \supset \mathbf{K}_{\sqrt{5}} \supset \mathbf{K}$. En pratique nous devons donc :

- soit construire la racine réelle $\sqrt{5}$ en appelant l'opération `all_roots_of` de $\{X \leftarrow \sqrt{5}\}_\mathbf{K}$ pour obtenir l'extension $\mathbf{K}_{\sqrt{5}}$.
  Nous devons ensuite construire la racine réelle α en appelant l'opération `all_roots_of` de $\{X \leftarrow \alpha\}_{\mathbf{K}_{\sqrt{5}}}$ pour obtenir finalement $\mathbf{K}_{\alpha,\sqrt{5}}$.
- Soit construire la racine réelle α en appelant l'opération `all_roots_of` de $\{X \leftarrow \alpha\}_\mathbf{K}$ pour obtenir l'extension $\mathbf{K}_\alpha$.
  Il nous faut ensuite construire la racine réelle $\sqrt{5}$ en appelant l'opération `all_roots_of` de $\{X \leftarrow \sqrt{5}\}_{\mathbf{K}_\alpha}$ pour obtenir finalement $\mathbf{K}_{\sqrt{5},\alpha}$.

Évidemment toutes ces extensions désignent le même corps et $\mathbf{K}_{\sqrt{5},\alpha} = \mathbf{K}_{\alpha,\sqrt{5}} = \mathbf{K}_{\sqrt{5}} = \mathbf{K}_\alpha$. Mais pour le démontrer il faut soit factoriser le polynôme $X^2 - 5 = (X - 2\alpha + 1)(X + 2\alpha - 1)$ dans $\mathbf{K}_\alpha$ soit factoriser le polynôme $X^2 - X - 1 = (X - \frac{1-\sqrt{5}}{2})(X - \frac{1+\sqrt{5}}{2})$ dans $\mathbf{K}_{\sqrt{5}}$.

Une solution possible serait d'autoriser le programmeur à décrire statiquement une tour d'extension suffisante pour qu'il y effectue tous ses calculs. On peut imaginer de factoriser les polynômes qui interviennent comme dans [Lan90]. Cette factorisation est coûteuse mais ne se fait qu'une fois. Cette solution a encore l'inconvénient de ne pas permettre la construction dynamique de nouveaux nombres.

En pratique nous préférons ne pas rechercher systématiquement la tour d'extension « la plus simple ». Nous nous contentons de nous « apercevoir » d'une simplification lorsque celle ci peut avoir lieu. Notre but est de faire en sorte que le corps de nombres algébriques que nous voulons implémenter puisse contenir un nombre arbitraire de variables algébriques.

Voyons comment établir la relation (2.1). Plaçons nous par exemple dans $\mathbf{K}_{\alpha,\sqrt{5}}$. La question qui se pose est de savoir si $2X = 1 + \sqrt{5}$ quand on interprète X valant $X \leftarrow \alpha$ dans $\{X \leftarrow \alpha\}_{\mathbf{K}_{\sqrt{5}}}$. On doit donc tester à zéro le polynôme $Q(X) = 2X - 1 - \sqrt{5}$ de $\mathbf{K}_{\sqrt{5}}[X]$ en $X \leftarrow \alpha$. Lors du calcul de pgcd de $Q(X)$ et de $P_\alpha(X) = X^2 - X - 1$ on voit que $Q(X)$ divise $P_\alpha(X)$ comme de plus $Q(\alpha)$ s'annule dans $\tilde{\mathbf{K}}$. On voit que

$$X - \frac{1+\sqrt{5}}{2}$$

est un meilleur polynôme de définition de α. On aperçoit donc que $Q(X)$ est nul et que l'égalité (2.1) est vraie.

2.3.3 Un exemple en FoC

Nous ne pouvons évidemment pas donner tout le code de la clôture réelle. Nous renvoyons à [Rio] pour le code Axiom. Un stage de DEA récent à montré qu'on pouvait en faire une implémentation en FoC. Nous en donnons ici une variation qui respecte le modèle présenté ici.

En FoC nous devons d'abord expliciter les types de données que nous utilisons pour :

1. représenter les éléments de $\mathbf{K}[X]$, les polynômes en une variable sur un corps $\mathbf{K}$ dont les éléments sont représentés par un type quelconque `'a` :

   ```
   type univ_poly_rep('a) = list(int*'a) ;;
   ```

   nous utilisons ici explicitement une représentation creuse. On pourra ensuite construire l'espèce FoC :

   ```
   species univ_poly(k is corps_ordonne)
     inherits polynomes_formels_univaries(k) =

     rep = univ_poly_rep(k) ;
     end
   ```

2. Les éléments de $\{X \leftarrow \alpha\}_\mathbf{K}$, le codage des racines réelles sont eux représentés par le type :

   ```
   type interval_coding_rep('a) =
     ('a*'a) * univpoly_rep('a) ;;
   ```

   nous utilisons ici le codage par intervalles avec un couple de points et un polynôme de définition. Notons que pour expliciter la représentation des éléments de $\{X \leftarrow \alpha\}_\mathbf{K}$ en fonction de la représentation `'a` des éléments de $\mathbf{K}$ nous avons été obligés d'expliciter la représentation des éléments de $\mathbf{K}[X]$ en fonction de `'a`. Nous reviendrons dans le chapitre 5 sur cette question.

   Nous pourrons ensuite avoir une espèce FoC :

   ```
   species interval_coding
      (k is corps_ordonne, k_x is univ_poly(k))

      inherits racine_reelle(k, k_x) =

      rep = interval_coding_rep(k) ;
   ```

3. Les nombres algébriques réels eux-mêmes sont des éléments de $\tilde{\mathbf{K}}$ dont la représentation peut être explicitée par :

   ```
   type alg_nums_rep('a) =
     | Base of 'a -> alg_nums_rep('a)
     | Composed of int ->
                 interval_coding_rep(alg_nums_rep('a)) ->
                 univ_poly_rep(alg_nums_rep('a)) ->
                 alg_nums_rep('a) ;;
   ```

ici bien sûr nous avons été aussi obligés de « déplier » complètement la représentation `alg_nums_rep` des éléments de $\tilde{\mathbf{K}}$ en fonction de celle (`'a`) des éléments de $\mathbf{K}$. Nous avons donc explicité les représentations de $\mathbf{K}[X]$ et de $\{X \leftarrow \alpha\}_{\mathbf{K}}$.

Nous pouvons maintenant implanter les nombres algébriques réels dans l'espèce FoC `nombres_algebriques_reels` en utilisant la structure de données `alg_nums_rep` avec comme paramètre la représentation pour un corps ordonné quelconque `k` :

```
species nombres_algebriques_reels (k is corps_ordonne)

  inherits corps_ordonne =

  rep = alg_nums_rep(k)

  let
  rec up_rec is polynomes_formels_univaries(self) =
              polynomes_univaries(self)
  and up_cod is racine_reelle(self,up_rec) =
              interval_coding(self)
  and plus = ...
```

Nous pouvons maintenant utiliser les opérations de

`polynomes_formels_univaries`

en les préfixant par `!up_rec` et celles de `racine_reelle` en les préfixant par `!up_cod`. Voyons par exemple comment implanter le calcul du signe d'un élément x de la clôture réelle $\tilde{\mathbf{K}}$ (`self`) de $\mathbf{K}$ (`k`) :

```
  and sign(x) = match x with
```

```
    | #Base(b) -> #Base(k!sign(b))
    | #Composed(age,code,val) ->
      let new_code = !intcod!sign(val,code) in
        match new_code with
        | #Res(res) -> res
        | #New(new_code) -> #foc_error("sign: bad call")
    end
```

Le signe d'un élément de base est celui qu'il a dans le corps de base. Pour le signe d'un élément composé nous appelons l'opération sign du domaine de codage. Cet appel doit retourner un résultat et non un nouveau codage. En effet comme dans la section 2.3.1 nous supposons que toutes les opérations produisent des polynômes premiers avec les polynômes de définition.

De même la fonction recip qui calcule l'inverse doit appeler l'opération recip correspondante qui doit inverser une expression en une variable au dessus d'un codage d'un nombre algébrique.

```
and recip(x) = match x with
    | #Base(b) ->
      let res = k!recip(b) in
      if #is_failed(res)
      then #Failed
      else #Unfailed(#Base(r))
    | #Composed(age,code,val) ->
      let new_code = !intcod!recip(val,code) in
        match new_code with
        | #Res(res) -> res
        | #New(new_code) -> #foc_error("recip: bad call")
    end
```

Pour les opérations binaires nous devons procéder de la même manière mais en déstructurant les deux arguments et en nous assurant que les expressions ont une forme correcte.

```
and plus(x,y) =  match x with
      | #Base(a) ->
      match y with
        | #Base(b) -> #Base(k!plus(a,b))
        | #Composed(ay,cy,vy) ->
          let s = !up_rec!plus(!up_rec!lift(x),vy) in
          !simplify(s, ay, cy)
```

```
          end
        | #Composed(ax,cx,vx) ->
        match y with
          | #Base(b) ->
            let s = !up_rec!plus(vx,!up_rec!lift(y)) in
            !simplify(s, ax, cx)
          | #Composed(ay,cy,vy) ->
            if #int_lt(ax,ay)(* cx created before cy *)
            then
              let s = !up_rec!plus(!up_rec!lift(x),vy) in
              !simplify(s, ay, cy)
            else
              if #int_lt(ay,ax) (* cy created before cx *)
              then
                let s = !up_rec!plus(vx,!up_rec!lift(y)) in
                !simplify(s, ax, cx)
              else (* ax=ay => cx=cy *)
                let s = !up_rec!plus(vx,vy) in
                !simplify(s, ax, cx)
          end
        end
```

Le rôle de la fonction `simplify` est de s'assurer que l'on ne puisse produire que des expressions dont la valeur polynomiale est première avec le polynôme de définition du codage. Lorsque le test à zéro retourne un nouveau codage, nous sommes sûrs que le polynôme de définition de la racine est soit un diviseur du polynôme passé en argument, soit un polynôme premier avec le polynôme passé en argument.

```
  and simplify(pol in up_rec, age in int, code in intcod)
        in self =
      if !up_rec!degre(pol) = 0
      then !up_rec!coefficient_dominant(pol)
      else
        let new_code = !upcod!is_zero(pol,code) in
        match new_code with
          | #Res(res) ->
            if res
            then #foc_error("prime computed zero")
            else #Composed(age, code, pol)
          | #New(new_code) ->
```

```
    let new_pol = !upcod!reduce(pol,new_code) in
    if !up_rec!degre(new_pol) = 0
    then !up_rec!coefficient_dominant(new_pol)
    else #Composed(age, new_code, new_pol)
  end
```

Chapitre 3

Les sous résultants

Nous avons vu qu'en pratique les algorithmes que nous avons utilisés pour la clôture réelle reposent sur des calculs de pgcd, de coefficients de Bezout ou de suites de Sturm. Tous ces algorithmes comme celui de calcul naïf du résultant sont des variantes de l'algorithme d'Euclide de calcul du pgcd.

3.1 Les sous résultants

Le problème de l'algorithme naïf de calcul du résultant (voir l'algorithme 1 de la section 1.2) est sa mauvaise complexité. En effet, d'après la formule matricielle pour le déterminant de Sylvester, le résultant est une fonction polynomiale des coefficients des polynômes P et Q d'entrée. En particulier il reste dans l'anneau des coefficients.

Son calcul explicite nécessite par contre de calculer le reste R de la division euclidienne de P par Q, ce qui ne peut se faire que dans un anneau où le coefficient dominant de Q est inversible. En pratique on se place donc dans le corps des fractions de l'anneau des coefficients que l'on suppose intègre évidemment.

Collins et Brown (voir [Loo83b, Col67, BT71]) ont indépendamment montré que l'on pouvait faire des calculs plus simples en restant dans l'anneau des coefficients de P et de Q. On remplace l'étape de division euclidienne par une étape de pseudo division et on peut faire des simplifications supplémentaires en divisant le résultat de la pseudo division par une quantité connue.

Algorithme 2 (Sous Résultants)
remplaçons dans l'algorithme 1 de la section 1.2 le reste

$$\mathtt{rem}(P,Q) = \frac{\mathtt{prem}(P,Q)}{\mathrm{lc}(Q)^{|P|-|Q|+1}}$$

par

$$\frac{\texttt{prem}(P,Q)}{\alpha\psi^{|P|-|Q|}}$$

et le paramètre λ par deux paramètres α et ψ initialisés à 1. On obtient l'algorithme des sous résultants :

$$\begin{array}{l}
\texttt{let resultant}(P,Q,\alpha,\psi) = \\
\quad Q = 0 \Rightarrow \psi \\
\quad R \leftarrow \frac{\texttt{prem}(P,Q)}{\alpha\psi^{|P|-|Q|}} \\
\quad \psi' \leftarrow \frac{\text{lc}(Q)^{|P|-|Q|}}{\psi^{|P|-|Q|-1}} \\
\quad \texttt{resultant}(Q,R,\text{lc}(Q),\psi')
\end{array}$$

et tous les coefficients qui interviennent restent dans l'anneau. C'est à dire que toutes les divisions sont exactes.

Algorithme 3 (Résultant général)
On peut exprimer un algorithme général de réduction en termes de 3 fonctions qui permettent de :

- calculer le prochain polynôme `next_defective`. Cette fonction doit faire décroître les degrés, ce qui garantit la terminaison.
- Calculer un polynôme similaire à celui par lequel on va diviser. Cette opération `next_non_defective` permet de calculer effectivement le résultant en fonction des polynômes calculés.
- Calculer un terme complémentaire pour simplifier le pseudo reste. Cette opération `next_alpha` trouve un diviseur du pseudo reste.

On obtient alors :

$$\begin{array}{l}
\texttt{let general_resultant}(P,Q,\alpha,\psi) = \\
\quad Q = 0 \;\Rightarrow \\
\quad\; |\,P\,| > 0 \Rightarrow 0 \\
\quad\; \psi \\
\quad Q' \leftarrow \texttt{next_non_defective}(Q,\psi,|\,P\,|-|\,Q\,|) \\
\quad \psi \leftarrow \text{lc}(Q') \\
\quad \alpha \leftarrow \texttt{next_alpha}(\text{lc}(Q)) \\
\quad R \leftarrow \texttt{next_defective}P,Q,\alpha,\psi) \\
\quad \texttt{general_resultant}(Q,R,\alpha,\psi)
\end{array}$$

L'algorithme naïf 1 de la section 1.2 peut alors être vu comme une instance de l'algorithme général en prenant $\alpha = \psi = \text{lc}(Q)$.

L'algorithme 2 habituel est aussi une instance de l'algorithme précédent en prenant

$$\begin{aligned}
&\texttt{next_alpha}(q) = q\\
&\texttt{next_non_defective}(Q,\psi,\delta) = \tfrac{-(-\mathrm{lc}(Q))^{\delta-1}Q}{\psi^{\delta-1}}\\
&\texttt{next_defective}(P,Q,\alpha,\psi) = \tfrac{\texttt{prem}(P,Q)}{-\alpha\psi^{|P|-|Q|}}
\end{aligned}$$

3.2 Les sous résultants faibles

Du point de vue de la clôture réelle nous sommes moins intéressés par le calcul du résultant que par des algorithmes de calcul de pgcd, de calcul des coefficients de Bezout ou de suites de Sylvester. Nous pouvons bien sûr modifier l'algorithme 3 pour retourner un pseudo diviseur de ses arguments ou bien l'étendre de façon classique pour calculer les coefficients de Bezout.

Dans [MMR96] j'avais proposé avec Marc Moreno une généralisation de l'algorithme des sous résultants. Cette généralisation est encore aujourd'hui la base de la résolution triangulaire de systèmes d'équations polynomiale.

Algorithme 4
L'algorithme `newSubResGcd` de [MMR96] est une instanciation de l'algorithme général 3 en prenant

$$\begin{aligned}
&\texttt{nextAlpha}(q) = \mid\mid q \mid\mid_f\\
&\texttt{nextNonDefective}(Q,\psi,\delta) = \tfrac{-(-\mid\mid \mathrm{lc}(Q)\mid\mid_f)^{\delta-1}\overline{\mathrm{lc}(Q)}^f Q}{\psi^{\delta-1}}\\
&\texttt{nextDefective}(P,Q,\alpha,\psi) = \tfrac{\texttt{prem}(\overline{\mathrm{lc}(P)}^f P,\overline{\mathrm{lc}(Q)}^f Q)}{-\alpha\psi^{|P|-|Q|}}
\end{aligned}$$

et en initialisant α et ψ à 1. On prend $x \to \overline{x}^f$ et $x \to \mid\mid x \mid\mid_f$ valant respectivement $f(x)$ et $xf(x)$ pour une fonction f quelconque transformant un élément non nul de $\mathbf{A}$ en un autre élément non nul de $\mathbf{A}$.

Cette modification revient à multiplier à chaque étape par $\overline{\mathrm{lc}(Q)}^f$ le polynôme Q par lequel on divise. Marc Moreno et moi avions pu montrer que les termes calculés restaient dans l'anneau des coefficients.

Proposition 4
Soit $\mathbf{A}$ un anneau intègre. Soient P et Q des polynômes de $\mathbf{A}[X]$ et f une fonction de $\mathbf{A}^*$ dans $\mathbf{A}^*$. Soit $F_0,\dots F_n$ la suite de polynômes calculée au cours de l'exécution de l'algorithme 4. Les coefficients des F_i $(0 \leq i \leq n)$ restent dans $\mathbf{A}$.

L'intérêt de l'algorithme 4 vient de ce que l'opération d'inversion dans un corps y est remplacée par une opération de quasi inversion.

Pour un élément x de $\mathbf{A}$, le quasi inverse de s est un couple $(\overline{x}, || x ||)$ avec $x\overline{x} = || x ||$. Si $|| x ||$ est « plus simple » que x sans que $\overline{x}^f$ soit « trop compliqué » on obtient des gains importants en temps de calcul.

Plus précisément supposons que l'on ait un anneau intègre $\mathbf{R}$ dit de base. Nous voulons travailler dans un anneau $\mathbf{A}$ qui contient des éléments qui sont algébriques sur $\mathbf{R}$. Au lieu de se placer d'abord dans le corps des fractions $\mathbf{Q} = \mathbf{R}^{*-1}\mathbf{R}$ de $\mathbf{R}$ et de travailler dans l'extension de corps $\mathbf{K} = \mathbf{A}^{*-1}\mathbf{A}$ de $\mathbf{Q}$ on peut remarquer que $\mathbf{K} = \mathbf{R}^{*-1}\mathbf{A}$. C'est à dire que l'on peut garder les dénominateurs dans $\mathbf{R}$ si l'extension $\mathbf{K}$ est algébrique sur $\mathbf{Q}$.

Si on considère qu'un élément de $\mathbf{R}$ est « plus simple » qu'un élément de $\mathbf{A}$ on peut voir l'inverse y d'un élément $x = a/d$ de $\mathbf{K}$ comme une fraction $y = d\overline{a}/|| a ||$ où a et $\overline{a}$ sont dans $\mathbf{A}$ et où d et $|| a ||$ sont dans $\mathbf{R}$. Ceci permet de déduire l'opération d'inversion de l'opération de quasi inversion.

Prenons un exemple très simple, si on veut inverser le nombre algébrique $\sqrt{5} + \sqrt{3}$ on trouve son quasi inverse $\frac{\sqrt{5}-\sqrt{3}}{2}$, c'est le couple $(\sqrt{5} - \sqrt{3}, 2)$. Dans un modèle de corps le dénominateur 2 se retrouve dans chaque terme algébrique et $\frac{\sqrt{5}-\sqrt{3}}{2}$ est en fait codé par $(\frac{1}{2})\sqrt{5} \oplus (\frac{1}{2})\sqrt{3}$ et le dénominateur 2 se trouve distribué dans toute l'expression la faisant grossir inutilement.

Soit $\mathbf{K}$ une extension simple de $\mathbf{Q}$ par un polynôme irréductible P_α dont les coefficients sont dans $\mathbf{R}$. L'inversion d'une expression $Q(\alpha)$ est alors directement la relation de Bezout qui se calcule avec l'algorithme d'Euclide étendu. La quasi inversion est obtenue par l'algorithme des sous résultants étendu et retourne un couple $(\overline{Q}, || Q ||)$ où $\overline{Q} \in \mathbf{R}[X]$ et où $|| Q || \in \mathbf{R}$ est le résultant de P_α et de Q. Ce quasi inverse est obtenu grâce à la relation qui exprime le résultant en fonction des polynômes d'entrée

$$\overline{Q}Q + \overline{P_\alpha}P_\alpha = || Q ||$$

Le gain de temps vient de la meilleure complexité de l'algorithme des sous résultants.

De plus dans l'algorithme 4 nous n'avons pas introduit d'inversion puisque toutes les divisions sont exactes. La première division de 4 correspond ainsi à une division exacte d'un élément de $\mathbf{A}[X]$ par un élément de $\mathbf{R}$. Cette division se fait terme à terme, on fait donc une suite de divisions exactes d'éléments de $\mathbf{A}$ par des éléments de $\mathbf{R}$.

L'opération de quasi inversion dans $\mathbf{A}$ revient à prendre le couple $(1, x)$ lorsque l'élément x que l'on veut inverser est dans l'anneau de base $\mathbf{R}$. Dans un cas plus compliqué où on travaille dans une tour d'extension $\mathbf{A}$ on fait toujours tourner l'algorithme 2 avec $f(x) = \overline{x}$. Comme $xf(x) = || x ||$ les divisions sont des divisions d'éléments de $\mathbf{A}$ par des éléments de $\mathbf{R}$ comme dans la deuxième division de l'algorithme 4.

FIG. 3.1 – Un nouveau modèle de clôture

$$\begin{array}{ccccc}
\mathbf{R} & \rightarrow & \mathbf{A} & \rightarrow & \tilde{\mathbf{R}} \\
\downarrow & & \downarrow & & \downarrow \\
\mathbf{R}^{*^{-1}}\mathbf{R} & \rightarrow & \mathbf{R}^{*^{-1}}\mathbf{A} & \rightarrow & \mathbf{R}^{*^{-1}}\tilde{\mathbf{R}} \\
\| & & \| & & \| \\
\mathbf{Q} & \rightarrow & \mathbf{K} & \rightarrow & \tilde{\mathbf{Q}}
\end{array}$$

En pratique, Nous devons simplement mieux gérer les dénominateurs. Le modèle pour les ensembles triangulaires implanté par Marc Moreno est de considérer l'anneau de base $\mathbf{R}$ comme plongé dans son corps des fractions $\mathbf{Q}$ et d'implanter les polynômes de $\mathbf{Q}[X_1, \ldots X_n]$ comme des couples formés d'un élément de $\mathbf{R}[X_1, \ldots X_n]$ et d'un élément de $\mathbf{R}$ en gérant un seul dénominateur par polynôme.

Dans ce modèle le plongement du corps $\mathbf{Q}$ dans un corps algébriquement clos ou réel clos reste implicite. Le nombre de variables est connu et fixé à l'avance et la résolution d'équation consiste à fournir un ensemble de systèmes triangulaires (voir [ALM99]) qu'il faut ensuite manipuler. Dans le cadre de la clôture réelle nous devons expliciter le corps dans lequel l'utilisateur travaille puisqu'on le lui fournit. On ne peut donc pas se contenter de simplement mieux gérer les polynômes dans la cadre de la clôture réelle.

3.3 Un modèle d'anneaux

Notre problème était donc de généraliser la construction de la clôture réelle d'un corps réel clos aux anneaux. Pour un anneau de base $\mathbf{R}$, nous devons construire un anneau $\tilde{\mathbf{R}}$ comme dans la figure 3.1 qui généralise le modèle des corps. Nous devons donner un modèle algébrique pour l'anneau $\mathbf{A}$ ainsi qu'un modèle pour les racines réelles $\{X \leftarrow \alpha\}_{\mathbf{A}}$.

Le cadre mathématique habituel est celui des *entiers algébriques* (voir [Lan64]) où on considère l'anneau des entiers $\mathbf{A}$ d'une extension algébrique $\mathbf{K}$ de $\mathbf{Q} = \mathbf{R}^{*^{-1}}\mathbf{R}$. Cet anneau est formé des éléments de $\mathbf{K}$ qui annulent un polynôme unitaire à coefficients dans $\mathbf{R}$.

D'un point de vue pratique nous voulons donc supprimer l'opération d'inversion d'un corps. Nous la remplaçons par l'opération de quasi inversion et une opération de division exacte. Celle ci doit prendre en arguments un élément d'un anneau « algébrique » $\mathbf{A}$ et un élément d'un anneau « de base » $\mathbf{R}$.

Dans le modèle de la section 2, nous avons deux opérations d'inversion `recip`. L'une $\mathbf{K} \to \mathbf{K}$ est celle des corps, l'autre $\mathbf{K}[X] \to \{X \leftarrow \alpha\}_{\mathbf{K}} \to \mathbf{K}[X]$ est celle des racines réelles que nous remplaçons par une opération `quasi_recip` de $\mathbf{A}[X]$ dans $\mathbf{A}[X] \times \mathbf{A}$. L'opération de corps est plus délicate à généraliser puisqu'on doit examiner en détail où apparaissent les dénominateurs dans l'ensemble des algorithmes qui sont utilisés.

Le schéma que nous avons finalement retenu est de modéliser ce qu'est une extension algébrique $\mathbf{A}$ d'un anneau $\mathbf{R}$ avec deux opérations

1. `quasi_recip` qui prend un élément de $\mathbf{A}$ et retourne une couple de $\mathbf{A} \times \mathbf{R}$. Ainsi si `quasi_recip`$(a) = (b, d)$ on a $ab = d$ mais d est dans $\mathbf{R}$.
2. `exquo` qui prend en arguments un élément de $\mathbf{A}$ et un élément de $\mathbf{R}$ et retourne un élément de $\mathbf{A}$. Ainsi si `exquo`$(a, d) = b$ on a $db = a$.

Tous les théorèmes du chapitre 1 s'appliquent sur des corps et leur généralisation à des anneaux n'est pas toujours possible. Par exemple pour isoler une racine d'un polynôme unitaire de $\mathbb{Z}[X]$, nous devons considérer que les bornes des intervalles sont rationnelles (soit dans $\mathbb{Q} = \mathbb{Z}^{*-1}\mathbb{Z}$). Par suite, l'évaluation d'un polynôme à coefficients entiers en des bornes rationnelles donne un rationnel. En pratique ceci nous amène à considérer une opération `local_eval` qui prenne en arguments un polynôme de $\mathbf{A}[X]$ et une « fraction » de $\mathbf{A} \times \mathbf{R}$ pour retourner une fraction de $\mathbf{A} \times \mathbf{R}$. Nous avons ainsi une « évaluation » `local_eval`$(P, \frac{a}{d_1}) = \frac{b}{d_2}$.

Même dans le cadre des entiers algébriques, nous montrons dans [Rio02] que l'on ne peut pas se retreindre aux polynômes unitaires de $\mathbf{A}[X]$ si on veut pouvoir simplifier des polynômes de définition. Le modèle de [Rio02] comprend donc trois niveaux de localisation

1. les fractions introduites par les bornes des intervalles,
2. les fractions introduites par la simplification de polynômes de définition des racines réelles,
3. les fractions globales qui permettent de travailler dans un corps réel clos.

3.4 Les suites de Sturm faibles

Dans le modèle à base d'entiers algébriques il nous manquait encore un moyen de généraliser les suites de Sylvester de la section 1.4.2. Il faut pouvoir continuer à faire le raisonnement local de la définition 1.4.1 mais la relation de proportionnalité change. Soit $F_0, F_1, \ldots F_n$ la suite des sous résultants de deux polynômes P et Q de $\mathbf{A}[X]$. On a la relation

$$\mathrm{lc}(F_i)^{\delta_i+1} F_{i-1} = K_i F_i + \alpha_i \psi_i^{\delta_i} F_{i+1} \tag{3.1}$$

où δ_i vaut $|F_{i-1}| - |F_i|$. On veut pouvoir contrôler les signes de façon à ce que $\mathrm{lc}(F_i)^{\delta_i+1}$ et $\alpha_i\psi_i^{\delta_i}$ soient de signe contraire.

La relation (3.1) est aussi celle de l'algorithme 4 et dans ce cas les coefficients α_i et ψ_i restaient dans l'anneau **R**. Nous avions donc d'abord généralisé en testant explicitement le signe de $\mathrm{lc}(F_i)$ dans l'anneau **A**. Cette technique avait l'inconvénient de calculer des signes (qui peuvent être compliqués à calculer) à chaque étape de réduction de l'algorithme 3 et, en pratique nous ne les utilisions que pour créer des racines réelles.

La modification que je propose dans [Rio02] permet de généraliser les suites de Sylvester sans test de signe dans l'anneau **A**. Ces tests sont simplement dans l'anneau **R** formé de nombres « simples » (des entiers en pratique). Ainsi la relation de proportionnalité entre les termes de $F_0, F_1, \dots F_n$ devient

$$|| \, \mathrm{lc}(F_i) \, ||_f^{\,\delta_i+1} F_{i-1} = K_i F_i - \alpha_i \psi_i^{\delta_i} F_{i+1} \tag{3.2}$$

où cette fois $|| \, \mathrm{lc}(F_i) \, ||_f$ reste dans **R**. On obtient alors la généralisation suivante :

Algorithme 5
Soit f un morphisme multiplicatif de $\mathbf{A}^*$ et soit g une fonction de $\mathbf{A}^*$ dans $\mathbf{R}^*$ retournant un diviseur de son argument. L'algorithme `quasi_subresultant` de [Rio02] s'obtient en prenant une instance de l'algorithme général 3 avec :

$$\begin{aligned}
&\texttt{nextAlpha}(q) = g(q)\\
&\texttt{nextDefective}(P,Q,\alpha,\psi) = \frac{-\texttt{prem}(P,\overline{\mathrm{lc}(Q)}^{f} Q)}{\alpha\psi^{|P|-|Q|}}\\
&\texttt{nextNonDefective}(P,\alpha,\psi,\delta) = \frac{(||\mathrm{lc}(Q)||_f)^{\delta-1}\overline{\mathrm{lc}(Q)}^{f} Q}{\psi^{\delta-1}}
\end{aligned}$$

et en initialisant α et ψ à 1.

On peut alors montrer que les termes calculés par cet algorithme restent dans $\mathbf{A}[X]$ et que les α_i et les ψ_i restent dans **R**.

Il est alors possible de se passer des méthodes basées sur la règle de Descartes (voir la proposition 3) qui ont l'inconvénient dans certains cas de devoir trop couper les intervalles de définitions des racines réelles. On provoque ainsi beaucoup de calculs de signes et d'évaluations de polynômes qui finissent par coûter cher.

L'algorithme 5 permet de contrôler les signes de la relation (3.2). On peut donc faire le même raisonnement que celui de la section 1.4.3 avec la suite de Sylvester *faible* calculée par l'algorithme 5 qu'avec la suite de Sylvester. On peut donc énoncer

Proposition 5
Soit $\tilde{\mathbf{Q}}$ un corps réel clos et $\mathbf{R}$ un sous anneau de $\tilde{\mathbf{Q}}$. Soit $\mathbf{Q} = \mathbf{R}^{*-1}\mathbf{R}$ son corps des fractions. Soit $\mathbf{A}$ un sous anneau de $\tilde{\mathbf{Q}}$ formé d'éléments entiers sur $\mathbf{R}$ et soit $\mathbf{K}$ le corps $\mathbf{R}^{*-1}\mathbf{A}$.

Soit $[a, b[$ un intervalle de $\tilde{\mathbf{Q}}$ avec a et b dans $\mathbf{Q}$. Soit P un polynôme sans facteurs carrés de $\mathbf{A}[X]$ tel que $P(b) \neq 0$ et $P(a).P(b) \leq 0$ dans $\mathbf{K}$. Soit α l'unique racine de P dans $\tilde{\mathbf{Q}}$. Soit Q un polynôme de $\mathbf{A}[X]$ premier avec P dans $\mathbf{K}[X]$.

Soit v_a et v_b les variations respectives de la suite de Sylvester faible de P et de Q en a et b alors
- si $P(b)$ est positif alors le signe de $Q(\alpha)$ dans $\tilde{\mathbf{Q}}$ est $v_a - v_b$.
- si $P(b)$ est négatif alors le signe de $Q(\alpha)$ dans $\tilde{\mathbf{Q}}$ est $v_b - v_a$.

Avec cette généralisation aux entiers algébriques, le nombre de calculs de signes que l'on doit faire est au plus deux fois le degré du polynôme de définition d'une racine réelle. Dans les méthodes basées sur la règle de Descartes le nombre de calculs de signes dépend de la distance euclidienne entre la position de la racine réelle et la valeur de l'expression dont on veut tester le signe. J'ai montré dans [Rio02] que le gain en temps de calcul pouvait être important lorsque les degrés des polynômes augmentent.

Conclusions

Mon problème était donc simple : chasser les dénominateurs des expressions contenant des nombres algébriques. J'ai été amené à concevoir un modèle qui sortait des catégories décrites en AXIOM pour pouvoir y revenir ensuite dans le cas des corps. Il fallait en effet spécifier la pseudo inversion et la division exacte externe afin d'en déduire l'inversion dans le cas des corps.

En pratique j'ai été amené à concevoir en même temps le modèle abstrait et son implémentation. Il me fallait concevoir un modèle algébrique différent qui n'utilise pas la division exacte `exquo` définie en AXIOM dans la catégorie `IntegralDomain`.

La clôture réelle représente 1000 lignes de code AXIOM. Il faut à peu près 700 lignes pour sortir du cadre des catégories standard AXIOM. On modélise ainsi la pseudo inversion et la division exacte externe. Une fois ce modèle écrit il faut encore 700 lignes pour implémenter les algorithmes du chapitre 3.

J'ai alors pu implémenter partiellement les algorithmes de la section 3.3 et la proposition 5. Le code fait alors 1200 lignes supplémentaires. On a donc fait plus que doubler la taille du code.

Il fallait donc recompiler 2500 lignes à chaque fois que je changeais le modèle de base pour mieux refléter les algorithmes. Je ne suis toujours pas satisfait du code AXIOM obtenu. Il est en effet très difficile de revenir dans la hiérarchie des catégories standard d'AXIOM. Là où j'attendais de la part du langage un support et une vérification de cohérence de l'ensemble du programme j'avais obtenu un code dont je ne savais pas toujours ce qu'il faisait.

Deuxième partie

FOC

Le projet FoC est directement issu des problèmes pratiques de compilation en Axiom ou Aldor. J'avais en effet de plus en plus de mal à « savoir » ce que faisait réellement le code que j'écrivais. Je me suis donc tourné vers les spécialistes de la sémantique des langages et de la preuve de programmes.

Nous avons décidé de démarrer le projet de concevoir et réaliser un « cadre » pour la programmation certifiée d'algorithmes de calcul formel. D'emblée nous avons écarté l'idée de redéfinir complètement un langage de programmation dédié au calcul formel. L'entreprise aurait été trop ambitieuse. Les efforts du projet FoC se sont concentrés sur l'aspect calculatoire qui constitue le « domaine d'expertise » du calcul formel. On ne peut pas certifier des programmes si on ne peut pas décrire ce qu'ils font c'est à dire leurs spécifications.

Au même moment, Loîc Pottier commençait à automatiser des mathématiques en Coq. Il voulait aborder les objets de base du calcul formel que sont les polynômes. Ses travaux sur l'algèbre en Coq ont mené à une collaboration entre le projet Lemme de l'INRIA Sophia et FoC au sein de l'action *Calcul Formel Certifié* (voir [Pot]).

En calcul formel, nos spécifications sont « les mathématiques du problème », il nous fallait donc un moyen de les décrire pour qu'elles deviennent explicites. On a vu dans la première partie que cela pouvait nous amener un peu loin lorsqu'on veut décrire des algorithmes non triviaux. Pour le chercheur en calcul formel, un nouvel algorithme n'a souvent d'intérêt que parce qu'il est plus rapide qu'un autre. Les mathématiques sous-jacentes ne nous intéressent que parce qu'elles permettent de justifier l'algorithme. Pour nous la preuve est un outil, certes important, mais ce n'est pas l'objet de nos recherches comme pour le chercheur de la théorie des preuves. Nous nous contentons souvent de « savoir » que l'on pourrait justifier le code qu'on a écrit.

Cette démarche est bien sûr insuffisante pour l'utilisateur de systèmes d'aide à la preuve. Il sait qu'une justification comme « l'anneau est nœtérien donc le calcul termine » est compliquée à mettre en œuvre. Nous voulions d'abord trouver un moyen de « structurer » les algorithmes et leurs spécifications avant d'attaquer leur certification. La preuve de programmes est toutefois bien présente dans le projet et le compilateur FoC actuel permet d'écrire des preuves en Coq.

Cette présentation ne parle pas du travail de sémantique du projet depuis cinq ans (voir [Pre00, Fechter01, PDH02, PD03]). Mon rôle à consisté à implanter des algorithmes de façon à ce que les sémanticiens puissent faire de la certification. Mais, avant de certifier les algorithmes, il faut pouvoir certifier le « modèle » que l'on utilise. Il ne sert à rien d'utiliser un trait de programmation si on ne peut pas en avoir la contrepartie dans un système de preuves. Par exemple les effets de bord sont compliqués à décrire dans les systèmes de preuves (voir [Fil99, Wad90]). Nous devions donc plutôt utiliser des fonctions récursives que des boucles.

Nous commencerons par préciser un certain nombres de notions classiques en théorie de la programmation. Nous ferons ensuite le lien entre ces notions et la programmation pour le calcul formel à travers différents prototypes d'une librairie d'algorithmes de base pour les polynômes. On pourra ensuite présenter une partie du langage FoC issu de ces prototypes. Nous présenterons enfin la librairie de polynômes que j'ai développée pour FoC.

Chapitre 4

Motivations

En calcul formel nous faisons des calculs exacts sur des éléments appartenant à des ensembles qui ont des structures algébriques parfois compliquées. Nous utilisons souvent des algorithmes sophistiqués manipulant des éléments qui occupent souvent une place importante en mémoire. Ainsi leur impression est parfois plus longue que leur calcul.

Comme les outils de calcul formel manipulent des formules mathématiques, ils sont souvent utilisés dans les phases de modélisation des projets. On attend d'eux la rigueur et la correction habituellement associées aux mathématiques. Beaucoup d'algorithmes de calcul formel reposent sur des théorèmes qui sont démontrés et ont donc une base solide. Malheureusement les programmes ne sont pas des simples recopies des théorèmes et même si les logiciels sont écrits avec soin une erreur reste possible.

Fréquemment un utilisateur de calcul formel augmente donc la confiance qu'il a sur le résultat fourni par le système en le vérifiant. Par exemple on peut souvent dériver une primitive pour retrouver le problème d'origine. Il arrive pourtant que l'utilisateur ne puisse pas du tout vérifier les résultats. Il doit alors faire confiance au système ou au moins à ceux qui ont écrit les parties sensibles des algorithmes mis en œuvre dans le système.

C'est donc souvent l'implémenteur de calcul formel qui garantit la correction et le bon fonctionnement de son programme. Pour l'assurer le programmeur doit avoir acquis un niveau de confiance « suffisant » dans le code qu'il a écrit. Il doit donc posséder une certaine maîtrise de la signification des constructions syntaxiques qu'il utilise.

Le programmeur n'utilise que les constructions qu'il « comprend bien » et dont il sait qu'elles correspondent à ce qu'il veut. Il utilise une discipline de programmation (les noms d'identificateurs, la structuration du programme) pour faire le lien entre les données qui sont manipulées par le programme et les objets abstraits qu'il a en tête. Il essaie souvent de « rapprocher » la formulation syntaxique d'une formulation mathématique. Il réduit ainsi la « distance » entre le code et les théorèmes sur lesquels repose sa cohérence.

Lorsque l'implémenteur a atteint un niveau de confiance suffisant c'est souvent parce qu'il s'est convaincu que le code écrit reflète bien les algorithmes. Comme ces derniers reposent sur des théorèmes mathématiques vérifiés, il a « confiance » en son code qu'il peut distribuer. Il peut rester des erreurs mais il sait que la base est solide. Il attend donc de la part du langage une aide pour exprimer les spécifications mathématiques de ses algorithmes.

4.1 Un exemple

Prenons mon exemple de portage de la clôture réelle d'Axiom à Aldor qui sont des langages extrêmement proches. Nous partions d'un programme existant en Axiom en lequel on avait une certaine confiance puisque de grosses applications (la décomposition cylindrique, la résolution triangulaire) l'utilisaient[1]. Nous devions changer de langage pour l'écrire en Aldor et utiliser la librairie BasicMath au lieu des libraires disponibles dans Axiom. Nous avons réalisé ce portage au sein de la société NAG qui commercialisait Axiom et Aldor et avons ainsi bénéficié des conseils des implémenteurs du langage et de la librairie.

En une journée, la traduction proprement dite était faite et nous avions un programme accepté par le compilateur que nous pouvions donc tester. Il nous a fallu une journée supplémentaire pour régler les problèmes liés à la traduction et obtenir un code qui fonctionne et soit capable de faire tourner sans erreurs la plupart des exemples d'Axiom. Au cours de cette phase nous avons détecté des problèmes sur les librairies de polynômes en une variable, le comportement de certaines fonctions n'était pas le même dans certains cas « dégénérés ». Nous avons aussi vu un problème plus grave sur une construction de base d'Aldor la fonctionnalité `extend` qui permet d'augmenter les opérations d'un domaine sans en changer le nom. Nous avons donc réécrit un certain nombre de fonctions mais un certain nombre de « petits problèmes » restaient en suspens. Certaines approximations n'étaient pas tout à fait les mêmes en Axiom et en Aldor ! Les approximations ne sont qu'un post-processing dans la clôture réelle et la partie purement calculatoire semblait

[1]en fait je n'ai eu vraiment « confiance » dans le programme que lorsque, à ma demande, Jean-Marie Arnaudiès est venu s'entretenir avec moi de nombres algébriques. Il m'avait, de mémoire, proposé certains de ses « exercices » destinés aux candidats à l'agrégation de mathématiques. Nous étions d'accord sur presque tout, sauf un exemple ... Chagriné, il est remonté dans son bureau ! Fort heureusement seule sa mémoire l'avait trahie et son livre était juste !

correcte[2].

Il nous a fallu une semaine pour identifier et corriger ces problèmes. Certaines opérations sur les polynômes modifiaient leurs arguments et lors de la création de nombres algébriques certains intervalles d'isolation étaient faux, cassant l'invariant sur lequel reposent tous les calculs d'approximation. Certaines redéfinitions d'opérations n'étaient pas prises en compte de la même manière qu'en AXIOM mais rien n'était spécifié dans les documentations. En conséquence certains calculs corrects en AXIOM étaient faux en ALDOR. Au cours de ces tests nous avons été amené à modifier nos sources et, en particulier, toutes nos fonctions qui faisaient des effets de bord. Il est resté un problème lié a ces modifications qui n'est apparu qu'un an plus tard ! Il s'agissait d'une fonction que nous avions instrumenté dans une phase de debogage de l'erreur. Elle ne correspondait plus à la version AXIOM.

4.2 Vers une maîtrise de la sémantique

Nous avons vu que pour pouvoir garantir la correction d'un programme nous devions avoir des assurances sur le comportement du langage source.

Dans un langage de programmation d'usage général (C par exemple) où les mathématiques restent implicites, on ne dispose que des constructions syntaxiques du langage pour implémenter nos algorithmes. Il y a donc naturellement une « distance » importante entre la formulation mathématique d'un algorithme et le programme qui l'implante.

Dans un langage dédié comme AXIOM, les spécifications mathématiques s'expriment dans le langage et on peut obtenir un meilleur niveau de confiance puisque la « distance » entre le code et les mathématiques sous-jacentes est plus faible. Un tel langage est plus expressif mais repose sur des concepts qui sont plus compliqués tels que les catégories ou les domaines. On doit alors se convaincre que, par exemple, la catégorie `Ring` en AXIOM est bien ce que l'on entend habituellement par un anneau en mathématiques.

Pour exprimer un algorithme en restant très proche des mathématiques les concepteurs d'AXIOM et d'ALDOR ont donc créé un langage offrant une

[2]l'importance des approximations ne peut pas être « négligée » par un programme qui manipule des nombres réels. Je ne pouvais pas m'imaginer qu'on puisse utiliser des nombres algébriques réels en dehors de la communauté de calcul formel et du calcul exact. Un jour le physicien Hubert Caprasse m'a proposé un système qu'il avait « résolu ». Il avait en effet exprimé les solutions en fonction des racines d'un polynôme en une variable. Il voulait la 78ème décimale de la solution, mais n'était pas « satisfait » des « résultats » de REDUCE. Évidement, lorsque l'on utilise des approximations dans les calculs, l'erreur grandit. Il fallait faire des calculs exacts et approcher à la fin. REDUCE, divergeait dès la 56ème décimale.

syntaxe et des constructions spécialement conçues pour les mathématiques. Si la syntaxe est relativement bien documentée, les constructions comme les catégories ou les domaines reposent sur des notions qui étaient très nouvelles en théorie de la programmation au moment des premiers compilateurs AXIOM. Ainsi, certains traits du langage ressemblent à des types abstraits algébriques, d'autres évoquent ceux des langages à objets.

La thèse de Guillaume Alexandre (voir [Ale98]), que j'avais partiellement encadré, avait porté sur la compréhension de la sémantique d'ALDOR. Il s'agissait de certifier en COQ des algorithmes écrits en ALDOR. Il avait choisi de décrire les catégories et les domaines d'AXIOM dans la théorie des ensembles de Zermelo, elle même exprimée en COQ.

Il a donc été naturellement conduit à modéliser les constructions de base d'AXIOM afin d'en avoir une vision certifiée en COQ. Il s'est aperçu que pour pouvoir faire la preuve d'une construction comme l'héritage des catégories (le `Join` d'AXIOM ou ALDOR) il était obligé de faire des choix sur l'ordre d'héritage. De même pour les domaines AXIOM il a dû se poser la question de savoir quelle était la portée des redéfinitions. Ces questions ne sont pas d'ordre mathématique, elles portaient sur la sémantique du langage.

On est donc conduit à se poser la question de la signification exacte de certains traits de programmation dans un cadre plus général que celui d'AXIOM ou d'ALDOR. Nous avons ainsi été amenés à réfléchir aux caractéristiques des langages appropriés pour le calcul formel.

4.3 Quelques traits des langages de programmation

Pour fixer le cadre de notre étude, rappelons quelques notions et paradigmes sous-jacents aux langages de programmation.

4.3.1 La programmation fonctionnelle

Classiquement, en mathématiques, une fonction est la donnée d'un ensemble de départ d'un ensemble d'arrivée et d'un graphe. Un langage de programmation doit offrir un moyen au programmeur d'implémenter cette fonction. La fonction mathématique sert alors de « spécification » au programmeur : implanter la fonction $\sin(x)$.

Du point de vue du calcul comme du point de vue algorithmique on doit distinguer plusieurs implémentations de la même fonction. Bien qu'elles calculent le même résultat à partir des mêmes arguments, elles n'ont pas obligatoirement le même code. On doit donc avoir une vision plus calculatoire qui considère une fonction comme un moyen de produire un résultat à partir de valeurs passées en argument. Le paradigme des *langages fonctionnels* considère les fonctions comme des valeurs que l'on nomme des *fermetures.* C'est cette vision λ-calcul des fonctions qui est généralement adoptée en sémantique pour parler des programmes.

On oppose souvent la programmation fonctionnelle à la *programmation impérative.* Dans une programmation impérative « à la Pascal » le programmeur met des valeurs dans des « boites » qu'il nomme et dont il modifie le contenu au cours de l'exécution du programme. Dans un langage purement fonctionnel ces variables n'existent pas et le programmeur doit uniquement utiliser des *identificateurs* pour nommer les valeurs qu'il manipule. Il décrit des expressions contenant des identificateurs qui prennent une valeur dans un certain contexte.

On a alors une *liaison* entre un identificateur et une valeur, autrement dit une association entre un identificateur et une valeur. Le contexte est un *environnement* que l'on peut voir comme une liste de liaisons dont les clefs sont les identificateurs. Un environnement apparaît donc comme un moyen de donner une valeur à une expression. Ainsi dans un environnement où le nom « `x` » vaut 1, l'expression `x+1` vaut la valeur 2.

Une fermeture est une valeur qui permet au programme de produire un résultat à partir d'arguments. Pour manipuler les arguments, le programmeur doit pouvoir les nommer dans l'expression qui forme le corps de la fonction qu'il écrit[3]. Elle a donc la forme syntaxique

$$(\texttt{fun X} \ \rightarrow \ \mathcal{E}_F)$$

où $\mathcal{E}_F$ est l'expression qui constitue le corps de la fonction et où `X` sont ses paramètres. En pratique $\mathcal{E}$ contient souvent les identificateurs `X` que le programmeur a utilisé pour nommer les arguments mais elle peut en contenir d'autres.

Pour simplifier supposons que `X` ne désigne qu'un seul paramètre. La valeur de l'*application* :

$$(\texttt{fun X} \ \rightarrow \ \mathcal{E}_F)(\mathcal{E})$$

dans un environnement E peut s'obtenir en substituant d'abord $\mathcal{E}$ aux occurrences de `X` dans $\mathcal{E}_F$. On obtient ainsi une expression $\mathcal{E}_n$ dont on prend la valeur dans l'environnement E. C'est la stratégie d'*appel par nom*, on l'oppose souvent à la stratégie d'*appel par valeur* qui consiste à évaluer d'abord $\mathcal{E}$ dans E. On obtient alors une valeur V que l'on substitue aux occurrences

[3]quoique certains langages utilisent des paramètres de position pour désigner le premier, le second argument ...

de `X` dans $\mathcal{E}_F$. Lorsque la stratégie d'appel par valeur termine elle est en général plus rapide et donne le même résultat que la stratégie d'appel par nom. Beaucoup de langages (C, OCAML, FOC) utilisent la stratégie d'appel par valeur.

Le corps $\mathcal{E}_F$ de la fermeture peut contenir des identificateurs libres. Ainsi (`fun x` $\rightarrow$ `x+y`) contient l'identificateur libre « `y` » qui doit avoir une valeur pour que l'application de cette fonction puisse produire un résultat. La valeur de « `y` » peut être prise dans l'environnement d'appel de la fonction. C'est la *portée dynamique* où on résout à chaque appel les liaisons. Cette stratégie peut être dangereuse dans la mesure où une redéfinition de « `y` » peut avoir des conséquences sur l'exécution de la fonction.

On oppose souvent la portée dynamique à la *portée statique* où la valeur de « `y` » est prise dans l'environnement de *définition* de la fermeture. Ainsi en portée statique les identificateurs sont liés aux valeurs qu'ils avaient au moment où la fonction a été définie. On peut alors faire des « optimisations » et remplacer (`fun x` $\rightarrow$ `x+y`) par (`fun x` $\rightarrow$ `x+1`) si « `y` » est lié à 1 au moment où on définit la fonction.

Une fermeture contient donc un moyen de lier les identificateurs à des valeurs en plus du code de la fonction elle même. Ainsi une fermeture est la donnée explicite d'identificateurs de paramètres, d'une expression constituant le corps de la fermeture et d'un environnement permettant de donner une valeur aux identificateurs libres du corps de la fermeture.

Soit F la fermeture qui est la valeur de l'expression (`fun X` $\rightarrow$ $\mathcal{E}_F$) dans un environnement (de définition) E_D. Pour calculer le résultat de l'application $F(X)$ dans un environnement (d'appel) E_A, il faut évaluer l'expression $\mathcal{E}_F$ dans l'environnement E_E (d'exécution) formé de E_D augmenté des liaisons (`X` , X) qui lient les noms de « `X` » aux valeurs de X dans E_A.

4.3.2 La surcharge

En mathématiques nous avons l'habitude de donner le même nom pour des objets utilisés de manière analogue. Ainsi les opérations arithmétiques usuelles (comme +, -, ...) ne désignent pas toujours la même opération. De même, certaines constantes usuelles (comme 0, 1 ...) ne désignent pas la même valeur. En informatique aussi les opérations arithmétiques sont souvent surchargées. Par exemple « `+` » peut désigner à la fois l'addition des entiers (`int`), celle des flottants (`float`) voire celles des `long` ou des `double`.

C'est le lecteur mathématicien qui donne alors un sens précis à la notation en s'appuyant sur le contexte d'utilisation de la notation. Par exemple si x et y sont dans $\mathbb{Z}$ et z dans $\mathbb{R}$, le $+$ de $y + z$ désigne l'addition dans $\mathbb{R}$ et la phrase « soit $x = y + z$ » n'est pas correcte dans ce contexte où x est entier.

En informatique, c'est le compilateur qui doit décider pour chaque occurrence d'un symbole surchargé quel est le code à exécuter. Par exemple l'addition des `int` et celle des `float` sont des instructions machine différentes. Un compilateur peut s'autoriser à prendre la partie entière d'un flottant pour en faire un entier. Par suite, en utilisant une syntaxe « à la C », le programme

```
int x,y ; float z;
x = y+z ;
```

n'a pas de signification claire. Doit on faire l'addition flottante et tronquer le résultat pour l'affecter à `x` , tronquer `z` et faire l'addition des entiers ou bien signaler une erreur ?

Il y a beaucoup de mécanismes pour désambiguïser la surcharge et ils sont souvent peu ou pas explicités ce qui peut être gênant pour le programmeur. L'introduction de surcharge fait que dans certains cas le compilateur ne peut pas décider quelle opération choisir. C'est le programmeur qui doit alors faire une déclaration ou désambiguïser « à la main » les expressions.

4.3.3 Le typage

Pour désambiguïser une expression contenant des opérateurs surchargés, nous sommes amenés à examiner le contexte dans lequel l'opérateur est utilisé. En mathématiques c'est le lecteur qui détermine le contexte et pose éventuellement la question si cela ne lui semble pas clair. C'est souvent en examinant l'ensemble dans lequel se trouvent les opérandes et le résultat que l'on donne un sens non ambigu a une expression. Le contexte pour un informaticien contient des informations provenant des déclarations des identificateurs. On peut les voir comme un environnement qui permet d'associer un type à un identificateur. Une telle « liaison » est une *signature* c'est à dire un couple formé d'un identificateur et d'un type. L'*inférence de type* consiste à associer un type à toute expression du langage comme l'évaluation consistait à lui associer une valeur. On l'oppose souvent à la *vérification de type* qui, étant donnés une expression et un type, montre que la valeur de l'expression dans un certain contexte a bien le bon type.

Les types de données

Les types de données jouent un rôle similaire à celui des ensembles en mathématiques car ils permettent de classifier les suites de bits qui sont manipulées par les processeurs. Ils donnent ainsi une *sémantique* aux données qui sont manipulées dans un programme. Lorsque le système de types que l'on utilise est un modèle des données circulant réellement dans un processeur (`int`, `float`), le typage permet de garantir qu'une instruction reçoit toujours des données ayant la bonne forme. Il est extrêmement important que par exemple, le compilateur n'appelle pas l'instruction d'addition des `int` avec deux opérandes qui sont des `long` de 64 bits. Le processeur prendrait le premier mot de 64 bits et ajouterait les deux moitiés, il pourrait ensuite prendre une partie du deuxième entier de 64 bits comme l'instruction suivante qu'il doit exécuter.

On a ainsi augmenté la sûreté d'exécution et on peut voir chaque expression d'un programme comme « appartenant » à un type.

Les types fournissent aussi une information de haut niveau utilisée par le compilateur pour produire du code exécutable correct. Ils permettent également au programmeur de mieux structurer ses données. Il peut alors donner une sémantique différente à une même donnée concrète. Ainsi un programme qui manipule des entiers modulo un entier n codé en un `int` peut les représenter en utilisant des `int` mais définir un type `modulo` différent de `int`. Ainsi la fonction d'addition des `int` ne s'appliquera plus à un `modulo`.

Les types abstraits

Un compilateur analyse l'ensemble des expressions que contient un programme. Il leur attribue un type, en utilisant des règles de base :

```
+ : int -> int -> int
```

par exemple. Il utilise également des règles supplémentaires qui viennent du typage des expressions précédentes. On peut imaginer de « grouper » des informations de typage dans une « interface » qui apparaît alors comme un ensemble de déclarations. On peut inclure ses signatures lors d'une compilation. Tout ensemble de valeurs respectant cette interface peut en être une implantation et être utilisé par un programme. Ce sont les modules simples d'Ocaml.

Ainsi un « domaine » Axiom apparaît comme un ensemble muni d'un certain nombre d'opérations. Elles sont spécifiées par la « catégorie » à laquelle il appartient. On peut voir ce domaine comme un type abstrait de données qui implante les opérations que l'on peut effectuer sur les valeurs du type. En Ocaml on écrirait par exemple :

```
module un_domaine =
stuct
  type D = un_type
  let op = une_op
end
```

pour désigner un domaine de nom « `D` » ayant une opération unaire de nom `op`. La représentation des éléments du domaine est alors le type `un_type` et l'opération est implantée à l'aide de `une_op` qui doit avoir le type

```
un_type -> un_type
```

La définition effective `un_type` pour les éléments de « `D` » est masquée comme si on avait écrit en OCAML :

```
module type un_domaine_visible =
stuct
  type D
  val op : D -> D
end
```

et contraint `un_domaine` à avoir le type `un_domaine_visible`.

Les catégories d'AXIOM offrent un mécanisme de niveau supérieur qui permet de décrire les opérations disponibles dans un domaine. Ce sont donc des types qui décrivent des types. Un domaine décrit par la catégorie se note « `%` » dans le code. En OCAML on peut imaginer d'écrire :

```
module type une_categorie =
stuct
  type %
  val op : (% -> %)
end
```

C'est ensuite le programmeur qui affirme qu'un domaine D « appartient » à une catégorie $\mathcal{C}$. En écrivant le code du domaine, il implante les opérations spécifiées par $\mathcal{C}$. La compilation produit un nouveau type D et la liste des signatures de $\mathcal{C}$ dans lesquelles « `%` » a été substitué à D. Il crée ainsi un nouveau type D et les opérations qui lui sont associées.

sous-typage

De même que l'on peut par une déclaration de type masquer les types concrets d'un module, on peut aussi oublier certaines de ses définitions. Ainsi le domaine précédent aurait pu implanter une autre opération binaire `op_bis` par exemple. Ainsi

```
module un_autre_domaine =
stuct
  type D = un_type
```

```
  val op = une_op
  val opbis = une_op_bis
end
```

peut être contraint à avoir le type `une_categorie` mais il peut aussi être contraint à avoir le type de `une_autre_categorie` :

```
module type une_autre_categorie =
stuct
  type %
  val op : (% -> %)
  val une_op_bis : % -> % -> %
end
```

et `une_autre_categorie` est un *sous-type* de `une_categorie`. Comme en OCAML un sous-type est un type moins général qui implante donc plus d'opérations.

4.3.4 L'héritage

Lorsqu'en mathématiques on dit qu'un corps est un anneau dans lequel tout élément non nul est inversible, on sous-entend que tout ce qui s'applique à un anneau s'applique aussi à un corps. Pouvoir exprimer cela dans un langage de programmation permet donc de faire décroître la distance entre la spécification mathématique et le programme.

On peut voir la dépendance de la structure de corps en fonction de la structure d'anneau comme un héritage au sens des langages à objets. La structure de corps enrichit la structure d'anneau d'un certain nombre d'opérations comme la division par exemple.

En informatique la notion d'héritage est traditionnellement liée aux langages à base d'*objets*. Nous renvoyons à [Rém02] pour plus de détails sur la programmation par objets en général, nous rappelons pour l'instant simplement quelques caractéristiques.

Un objet possède des *méthodes* qui sont les opérations qu'il peut reconnaître ainsi que des *variables d'instance* qui caractérisent son état. On groupe les objets en *classes* qui partagent toutes les méthodes d'une famille d'objets. Ils sont distingués entre eux par les valeurs des variables d'instance. La programmation fonctionnelle classique qui consiste à programmer en appliquant une fonction sur des arguments est alors remplacée par l'envoi d'une méthode à un objet dans la programmation par objets. Les objets eux mêmes sont créés par l'intermédiaire des classes et grâce à une primitive « `new` ». Beaucoup de langages offrent la notion de classe « virtuelle » ou abstraite qui permet de spécifier des méthodes sans les définir. Ainsi `new` ne peut s'appliquer à ces classes.

Intuitivement on peut voir un objet d'une classe comme un élément d'un ensemble et les méthodes de la classe comme les opérations que les objets savent effectuer. Nous verrons dans le chapitre 5 les limitations de cette approche.

Les classes peuvent *hériter* d'autres classes et redéfinir certaines méthodes ou bien en définir de nouvelles. Une classe qui hérite d'une autre en est une *sous-classe*. Certains langages permettent un héritage simple, c'est à dire qu'une classe ne peut hériter que d'une seule classe. D'autres langages autorisent l'héritage multiple et une classe peut hériter de plusieurs classes.

Dans le premier cas, la relation d'héritage définit un arbre. Pour un objet d'une classe donnée, on peut trouver l'implémentation d'une méthode en remontant dans l'arbre d'héritage de la classe. Comme le chemin menant à la racine est unique et entièrement spécifié par la classe à laquelle l'objet appartient on peut donc résoudre l'héritage.

Pour les langages à héritage multiple la relation d'héritage n'est plus un arbre mais un graphe. On doit alors spécifier un ordre parmi les parents d'une classe pour pouvoir résoudre les noms de méthodes.

L'état d'un objet est en général caché à l'extérieur de cet objet. Les variables d'instances sont alors *privées*. Seules les méthodes sont *exportées* et visibles de l'extérieur de l'objet. L'*interface* d'un objet est donc constituée de l'ensemble de ses méthodes. L'interface d'une sous-classe est un sous-type de celles des classes dont elle hérite qui définissent moins de méthodes. Il se peut que les interfaces de deux classes soient en relation de sous-typage sans que les classes héritent. Elles peuvent en effet avoir des méthodes de même nom.

4.3.5 La liaison tardive

Dans le paradigme des objets les classes peuvent toujours redéfinir des méthodes qui sont héritées. Prenons l'exemple suivant qui utilise la syntaxe de FoC avec les notations OCaml pour l'arithmétique des entiers. Ici, une espèce est assimilable à une classe d'un langage à objets, une collection est assimilable à un objet que l'on produirait par `new`. L'appel de méthode se note « `!` », la définition d'une méthode se fait par `let` et `self` désigne l'objet modélisé par la classe que l'on définit. Ces notations sont celles de la section 5.2.

```
species pair_impair =
  let rec
```

```
      pair(x) =
        if (x=0)
        then true
        else self!impair(x-1)
  and impair(x) =
        if (x=0)
        then false
        else self!pair(x-1)
  ;
end ;;

collection pair_slow implements pair_impair = end ;;

species pair_only
  inherits pair_impair =
  let pair(x) = ((x mod 2) = 0) ;
end ;;

collection pair_fast implements pair_only = end ;;
```

Nous définissons ici deux méthodes `pair` et `impair`. Elles sont implantées dans la classe `pair_impair`. Nous définissons aussi un objet nommé `pair_slow` instanciant cette classe. La classe `pair_only` redéfinit la méthode `pair` et l'objet `pair_fast` en est une instance.

Dans `pair_impair` on donne une définition générale des fonctions `pair` et `impair` qui est correcte mais très inefficace. La classe `pair_only` donne une meilleure définition pour la méthode `pair` mais hérite de l'implantation de `impair` donnée dans `pair_impair`. Lors de la création de l'objet `pair_slow` on a deux fermetures P_{slow} et I_{slow} auxquelles sont liés les noms de méthode `pair` et `impair`. L'environnement de définition de ces fermetures contient une liaison du nom `pair` à la valeur P_{slow} et une liaison du nom `impair` à la valeur I_{slow}. Par contre lors de la création de l'objet `pair_fast` on a deux fermetures P_{fast} et I_{fast} auxquelles sont liés les noms `pair` et `impair`. L'environnement de définition de I_{fast} contient la liaison du nom `pair` à la valeur P_{fast}. On a donc un bon comportement de la méthode `impair` puisqu'elle se contente d'appeler la méthode `pair` qui est efficace.

Si la liaison était statique la redéfinition du nom `pair` ne serait pas vue par `impair` et l'environnement de définition de I_{fast} serait le même que celui de I_{slow} qui lie donc le nom `pair` à la valeur P_{slow} provoquant l'appel de la méthode `pair` comme définie dans la collection `pair_slow`.

Chapitre 5

Le modèle FoC

Notre première tâche, dans la construction du modèle FoC, était de déterminer les traits de programmation nécessaires pour écrire des algorithmes de calcul formel. Au cours des réunions du projet nous nous sommes rapidement aperçus qu'il fallait utiliser un vocabulaire commun. Le mot « objet » était en particulier beaucoup trop surchargé ! En tant que programmeur AXIOM j'avais l'habitude d'utiliser les mots « catégories » et « domaines » qui n'avaient pas le même sens pour les autres membres.

Le lexique que nous avons retenu est de considérer qu'un système de calcul formel manipule des *entités* qui appartiennent à des *collections* dont les opérations sont décrites par des *espèces*. Informellement les entités sont les éléments, les collections sont les ensembles et les espèces sont les structures algébriques qui les décrivent. Ainsi, en première approximation, pour AXIOM une entité se nomme un élément, une collection se nomme un domaine et une espèce se nomme une catégorie.

5.1 Le codage

Notre problème est de savoir ce que doivent être les entités, les collections et les espèces dans un langage de programmation. Doit on redéfinir complètement un langage comme AXIOM ou ALDOR ? Il nous faut alors complètement expliciter ses constructions syntaxiques, sa sémantique, son mécanisme d'évaluation, son récupérateur de mémoire ... L'objectif de FoC est différent puisque l'on souhaite mettre plus l'accent sur la certification et permettre à un programmeur de justifier formellement ses algorithmes.

En ce qui concerne l'aspect langage, l'approche de FoC est pragmatique. On ne se pose pas la question des constructions d'un langage de programmation. Ainsi les structures de données habituelles en informatique comme les listes ou les tableaux sont considérées comme externes à FoC. Elles n'ont en effet pas grand chose à voir avec le calcul formel. La conception d'un langage permettant de manipuler efficacement des données informatiques et d'offrir des fonctionnalités de haut niveau au programmeur sort de notre domaine de compétence. Nous considérerons que les manipulations de listes, de tableaux ou d'autres structures de données sont connues. Le langage de programmation que nous utiliserons devra être capable de gérer cela efficacement pour nous.

Nous voulons nous concentrer sur l'aspect « calcul formel » en nous appuyant sur la possibilité qu'a un chercheur de calcul formel de formuler ses algorithmes de différentes façons. Nous voulions ainsi dégager des « règles » de programmation. Je savais bien qu'il était toujours possible de coder une librairie de calcul formel. Mon problème était donc plus un problème « de style » pour dégager les traits de haut niveau nécessaires à l'implémentation de mathématiques. Nous voulions donc « représenter » les espèces et les collections dans un langage de programmation.

5.1.1 Méthodologie

La première question qui se pose à l'automne 97 quand nous voulons « coder » ou « représenter » ces notions est celle du langage de programmation que l'on va utiliser. Notre choix s'est naturellement porté vers un langage offrant des concepts de haut niveau dans lequel on retrouve les grands traits de programmation de la section 4.3. Il nous fallait un langage avec une sémantique bien comprise afin de pouvoir en dégager un modèle. On ne voulait pas se contenter de programmes qui « fonctionnent ».

Le choix d'OCAML était donc raisonnable puisqu'il offre les notions d'objet (voir 4.3.4), de modules (voir 4.3.3) ... Nous avons donc été naturellement amenés à ne pas considérer les notions de surcharge (voir 4.3.2) afin de simplifier l'écriture de la librairie. On aurait probablement pu choisir le langage HASKELL mais il nous a semblé qu'en termes d'exécution OCAML offrait de meilleures perspectives et nous connaissions mieux OCAML. Une autre possibilité aurait été de se restreindre à un sous ensemble « bien compris » d'AXIOM mais les travaux de [Ale98] ou [PT00] nous en ont découragé.

Notre but est de dégager ce que sont vraiment les espèces et les collections. L'idée que nous en avons est suffisamment précise pour dire qu'elle doivent ressembler aux ensembles et aux structures mathématiques. Elle est par contre trop floue pour en donner directement une description précise, en termes de constructions syntaxiques. Nous avons donc choisi une approche pragmatique qui consiste à viser des objectifs déjà atteints par certains systèmes de calcul formel.

Pour décider d'une méthode d'implantation, nous avons choisi de mener quelques expériences avec les polynômes et leurs représentations classiquement utilisées en calcul formel. Nous nous somme fixé l'objectif de dégager une « définition » des espèces et des collections qui soit aussi générique que celles de catégorie et de domaine en AXIOM. C'est en effet AXIOM qui offre le plus d'expressivité au programmeur d'algorithmes mathématiques. Il peut y définir un « cadre mathématique » pour y exprimer ses algorithmes. Par exemple, je n'aurais pas pu décrire les algorithmes de la première partie dans un système comme MAGMA (qui n'existait de toute façon pas encore) où il est impossible de définir des « catégories ».

Nous reviendrons dans le chapitre 6 sur les algorithmes effectivement utilisés. Intéressons ici nous à la manière de développer une librairie c'est à dire à la structurations des unités de la librairie. Nous voulons évidemment rester aussi proche que possible des mathématiques que nous implantons. Informellement nos objectifs sont donc :

- pouvoir construire, par exemple, un groupe à partir d'un monoide.
- Reconnaître qu'un groupe est un monoide particulier.
- Pouvoir énoncer un algorithme le plus tôt possible. L'algorithme d'Euclide se code dans un anneau euclidien.
- Ces définitions « par défaut » d'algorithmes doivent pouvoir être redéfinies en prenant en compte de nouvelles opérations.

Nous avons choisi de ne pas refléter les propriétés dans la librairie OCAML. Le système de types d'OCAML est insuffisant pour les décrire. Toutefois nous voulions traduire une partie de la « dépendance mathématique » en OCAML. Nous ne voulions pas nous limiter à l'aspect calculatoire d'OCAML mais nous attendions qu'il nous facilite la description des mathématiques.

Nous avons donc choisi d'implanter plusieurs prototypes en utilisant différents « paradigmes ». J'ai mené ces développements en parallèle afin de mieux voir ce qui s'exprime mieux avec une approche plutôt qu'une autre.

Pour cette partie nous utiliserons la syntaxe d'OCAML, nous renvoyons à [LDG+00] pour plus de détails sur sa syntaxe et à [CM98] pour la programmation en OCAML.

5.1.2 Approche par module

En reprenant le paradigme d'AXIOM (voir 4.3.3), on voit qu'une catégorie apparaît comme une liste de signatures comportant des types et des opérations sur ces types. La notion semble assez proche de celles des modules d'OCAML et dans cette approche nous implémenterons un ensemble par un module OCAML. Une structure algébrique (comme un anneau) est alors un type de module. Par exemple un monoide exporte un type de données abstrait `rep` qui sera instancié par une représentation pour un monoide particulier comme les entiers. Une opération est spécifiée par son type comme dans « `plus : rep -> rep -> rep` ». Ainsi :

```
module type Monoide_additif =
sig
  type rep
  val zero : rep
  val plus : rep -> rep -> rep
end
```

définit un monoide additif que l'on peut réaliser par

```
module Entiers : Monoide_additif =
struct
  type rep = int
  let zero = 0
  let plus x y = x+y
end
```

Dans cet exemple, le type effectif `int` pour la représentation est masqué par la déclaration « `Entiers : Monoide_additif` » qui fournit un type abstrait. Du point de vue du typage le type `Entiers.rep` est distinct du type `int`.

La difficulté pratique vient du fait qu'il est difficile de développer rapidement en utilisant ce paradigme. Par exemple pour définir un groupe additif à partir d'un monoide additif il nous faut simuler l'héritage.

```
module type Groupe_additif =
struct
  module Papa : Monoide_additif
  val oppose : Papa.rep -> Papa.rep
  val moins : Papa.rep -> Papa.rep -> Papa.rep
end
```

la notation `Papa.rep` permet d'accéder au champ `rep` du module `Papa`. On peut aussi inclure automatiquement les signatures correspondant aux monoides additifs dans celles des groupes additifs

```
module type Groupe_additif =
struct
 type rep
```

```
  val zero : rep
  val plus : rep -> rep -> rep
  val oppose : rep
  val moins : rep -> rep -> rep
end
```

C'est le mécanisme de typage des modules d'OCAML (voir 4.3.3) qui se chargera de vérifier que le type de module `Monoide_additif` est bien compatible avec `Groupe_additif`.

Pour les besoins de l'exposé supposons que nous ayons une « macro » `include` qui nous permette d'inclure toutes les signatures d'un module. Dans ce formalisme un groupe additif se décrit alors

```
module type Groupe_additif =
struct
  include Monoide_additif
  val oppose : rep
end
```

On a ici écrit la dépendance entre un groupe et un monoide. La cohérence repose toutefois sur ce que fait réellement la macro `include`. La question est beaucoup plus difficile qu'il y parait. Les travaux de Jérôme Vouillon (voir [Vou98]) montrent que cette utilisation des modules est proche de la programmation objet. La sémantique de FOC que Sylvain Boulmé a fait en COQ dans sa thèse montre aussi que `include` ne peut pas être une macro et qu'il faut gérer la cohérence au niveau du compilateur.

Les modules OCAML peuvent être paramétrés par d'autres modules ce qui nous permet de traduire des dépendances mathématiques. Ainsi dans un groupe on sait que $x - y = x + (-y)$ et on voudrait pouvoir le traduire dans le code en donnant une définition « par défaut » à `moins`. Nous devons écrire un module

```
module Groupe_par_defaut(G : Groupe_additif) =
struct
  let moins x y = G.plus x (G.oppose y)
end
```

Cette définition par défaut peut être incluse par tout module implantant un groupe additif. Nous pouvons même déclarer ses signatures dans un autre module :

```
module type Groupe_additif_defaut(G : Groupe_additif) =
```

```
struct
  include Groupe_additif
  val moins : rep -> rep -> rep
end
```

Le paramétrage permet de décrire des structures plus compliquées comme les **K**-espaces vectoriels. Supposons avoir défini un type de module `Corps` nous pouvons écrire

```
module type Espace_vectoriel(K: Corps) =
struct
  include Groupe_additif
  val multiplication_scalaire : K.rep -> rep -> rep
end
```

qui définit la multiplication par un scalaire.

Avec cette approche nous avons du mal à considérer les structures quotient comme par exemple les entiers modulo n. Il nous faut pour cela prendre un singleton et accéder à sa valeur à travers un champ (`get_it` par exemple) d'un module. Par exemple

```
module type Singleton =
struct
  type t
  val get_it : t
end
```

Nous sommes ainsi obligés d'« internaliser » la valeur dans un module. Par exemple en supposant que « `n` » stocke l'entier n :

```
module Un_singleton : Singleton =
struct
  type t = int
  val get_it = n
end
```

Nous pouvons ensuite écrire les entiers modulo un entier n passant le module en argument :

```
module Modulo(N : Singleton with type t = int) :
  Groupe_additif =
struct
  type t = int
```

```
  val the_mod = N.get_it
  val plus x y =
    let res = x+y in
    res mod the_mod
  ...
end
```

la fonction OCAML `mod` retourne le reste de la division euclidienne de deux `int`.

Si nous voulons utiliser les définitions par défaut des groupes additifs nous pouvons créer un nouveau module qui les utilise :

```
module Modulo(N : Singleton with type t = int) :
       Groupe_additif_avec_defaut =
struct
  include Modulo(N)
  include Groupe_par_defaut(Modulo(N))
end
```

Nous voyons donc que ce paradigme de modules a plusieurs inconvénients

- il est difficile à mettre en œuvre sans avoir complètement explicité comment inclure un module dans un autre.
- Les opérations par défaut s'expriment mal et leur mise en œuvre est délicate.
- La paramétrisation par des valeurs est compliquée à utiliser. Un mathématicien s'attend à avoir la valeur de n directement disponible lorsqu'il parle de $\mathbb{Z}/n\mathbb{Z}$.

Nous avons donc rapidement abandonné cette approche.

5.1.3 Approche naïve

Les problèmes de la macro `include` nous conduisent à nous intéresser à la programmation par objets. On peut en effet penser que cette « macro » s'apparente à de l'héritage (voir la section 4.3.4).

Lorsqu'on « débute » avec la programmation objet, une classe apparaît comme un contenant pour les objets qu'elle permet de créer. On est donc tenté de faire l'assimilation entre une classe et un ensemble ainsi qu'entre un objet et un élément de la classe. Ce sont les valeurs prises par les variables d'instance qui distinguent les différents éléments de l'ensemble. C'est de cette manière que sont implantées beaucoup de notions mathématiques dans des langages comme JAVA ou C++. Dans cette approche on peut assimiler une classe virtuelle a une structure algébrique.

L'héritage des classes permet en effet de bien modéliser les dépendances mathématiques surtout en OCAML qui offre de l'héritage multiple. Examinons comment les méthodes sont typées sur l'exemple d'un monoide additif $\mathcal{M}$ possédant une opération binaire notée additivement. Ainsi « $+$ » est associative, commutative et possède un élément neutre noté 0. Dans notre paradigme nous avons donc deux méthodes `plus` et `zero` que l'on doit spécifier.

Comme les éléments des ensembles sont des objets, nous devons spécifier que `zero` a le type des objets de la classe soit « `'self` » en OCAML. Pour l'addition, nous devons envoyer la méthode à un objet de la classe, il faut donc qu'un élément de $\mathcal{M}$ « reçoive » la méthode. On peut interpréter cela comme un ordre d'addition envoyé à un élément a de $\mathcal{M}$ avec un autre élément b de $\mathcal{M}$.

En termes de syntaxe concrète on peut imaginer que l'expression « `a+b` » est transformée en « `a#plus b` ». La notation « `#` » désigne l'envoi de méthode en OCAML. La méthode `plus` doit donc prendre un objet de la classe en argument et retourner le résultat de l'addition qui est encore un objet de la classe. On obtient donc le type « `'self -> 'self` » pour la méthode `plus`. La classe virtuelle qui spécifie les opérations d'un monoide est donc :

```
class virtual monoide_additif =
object{self:'self)
  method virtual zero : 'self
  method virtual plus : 'self -> 'self
end
```

Si maintenant on veut décrire les groupes additifs on pourra hériter de cette classe ce qui construira bien la notion de groupe à partir de celle de monoide. Ainsi :

```
class virtual groupe_additif
object{self:'self)
  inherits monoide_additif
  method virtual oppose : 'self
  method moins y = self#plus (y#oppose)
end
```

Le type de la méthode `oppose` qui retourne l'inverse additif d'un élément du groupe est naturellement « `'self` ». Pour calculer la différence de deux éléments il faut ajouter au premier l'opposé du deuxième. Il faut donc envoyer la méthode `plus` au premier élément, en lui passant l'opposé du deuxième qui est le résultat de l'envoi de la méthode `oppose`. On obtient bien le type « `'self-> 'self` » comme type pour la méthode `moins`. Par rapport à l'approche à base de modules, l'héritage nous a permis de bien exprimer la dépendance de `plus`, `oppose` et `moins`. Il n'est pas nécessaire que les méthodes soient définies pour être utilisées.

On voit par contre que l'arité des méthodes n'est pas cohérente puisque `zero` et `oppose` ont la même arité. L'écriture du code et des opérations binaires est également assez loin de l'habitude mathématique. Par exemple, montrer que l'addition est commutative revient à dire que pour deux objets a et b de la classe, les résultats de a`#plus`(b) et celui de b`#plus`(a) sont les mêmes. On voit difficilement comment faire sans modéliser complètement l'envoi de méthode dans les langages orientés objet ce qui nous éloigne du calcul formel.

Comme pour le prototype à base de modules, imaginons que nous voulions implanter les entiers à l'aide des entiers de base d'OCAML. Nous pouvons imaginer de placer la valeur de l'entier que l'on souhaite coder dans une variable d'instance de nom `rep` et faire les opérations nécessaires.

On code ainsi 0 en créant un objet dont le champ `rep` vaut l'entier OCAML 0. Comme la méthode est envoyée à un objet déjà existant, celui ci n'a qu'à se cloner en mettant 0 dans la variable d'instance `rep`.

```
class entier(n) =
object(self:'self)
  inherits monoide_additif
  val rep = n

  method zero = {< rep = 0 >}
```

Le paramètre `n` ne sert que d'initialiseur. La syntaxe « `{< rep = 0 >}` » duplique l'objet avec la valeur 0 pour la variable d'instance `rep`.

Pour l'addition il nous faut retourner un objet contenant le résultat dans son champ `rep`. C'est la somme de la valeur du champ `rep` de l'objet a qui on a envoyé la méthode et de la valeur du champ `rep` de l'objet passé en argument. Dans les langages à base d'objets les variables d'instances sont privées. Comme elles ne sont pas accessibles par d'autres objets il nous faut exporter le champ `rep` dans une méthode `get_rep` par exemple. On a donc

```
  method get_rep = rep

  method plus other = {< rep = rep + other#get_rep >}
end
```

On voit ici que l'on doit en permanence encapsuler les entiers que l'on veut coder dans une variable d'instance et créer beaucoup de nouveaux objets. Le modèles est donc assez lourd mais, d'après les essais que nous avons fait, ceci n'est pénalisant que pour des petites données. Dès que celles ci sont plus grosses, le surcoût est moins important.

Nous sommes par contre assez loin d'une formalisation proche des mathématiques puisque nous avons dû sacrifier l'arité des fonctions. De plus la représentation est devenue explicite et accessible depuis n'importe quelle fonction qui connaît la classe. Nous avons toutefois pu aller jusqu'à la programmation des polynômes dans ce modèle.

Examinons la question de la paramétrisation et imaginons que nous ayons une classe virtuelle `corps` qui décrive les corps. Nous voulons encore une fois décrire les espaces vectoriels sur un corps $\mathbf{K}$. Les classes OCAML peuvent recevoir des paramètres de type ainsi que des paramètres de valeur. Le corps $\mathbf{K}$ doit être implanté dans une classe qui n'est pas une valeur normale en OCAML. On doit donc identifier le corps $\mathbf{K}$ avec un type de classe OCAML, nous le désignerons par « `'k` » dans le code.

Nous voulons pouvoir exprimer que le paramètre est un corps et écrire une contrainte de type en OCAML que le compilateur OCAML puisse vérifier et accepter.

Si $\mathbf{V}$ est un $\mathbf{K}$-espace vectoriel, la multiplication par un scalaire est une fonction de $\mathbf{K} \times \mathbf{V}$ dans $\mathbf{V}$. Pour λ dans $\mathbf{K}$ et v dans $\mathbf{V}$, nous interprétons $\lambda.v$ comme l'envoi de la méthode « . » à l'objet v qui retourne donc λv. On est donc amené à écrire

```
class virtual [ 'k ] espace_vectoriel =
object(self : 'v)
  constraint 'k = #corps
  inherits groupe_additif
  method virtual multiplication_scalaire : 'k -> 'v
end
```

Le type « `#corps` » désigne le type de tous les objets qui sont compatibles avec la classe `corps`. La technique est un peu compliquée mais on arrive à décrire cette dépendance. Du point de vue de l'exécution du code, cette clause ne sert à rien. Elle permet néanmoins d'affirmer l'usage mathématique qui veut qu'on ne parle d'espaces vectoriels que sur un corps. Le code est ainsi « plus proche » des mathématiques. Notre but n'était pas seulement d'avoir un prototype qui fonctionne mais qui puisse aussi refléter ce que le mathématicien « dit ».

La paramétrisation par des valeurs ne pose par contre pas de problèmes comme dans l'approche à base de modules. On peut écrire directement les entiers modulo n en paramétrant par l'objet « `n` » qui est bien une valeur OCAML :

```
class ['integer] entiers_modulo(n : 'integer)
object(self)

  constraint 'integer = #entiers

  inherits groupe_additif
  val rep = 0
  method zero = {< rep = 0 >}
  method get_rep = rep
  method plus(other) =
   let res = rep + other#get_rep in
   {< rep = res mod (n#get_rep) >}

  method oppose = {< rep = (n#get_rep) - rep>}
end
```

Ici nous avons explicité le type « `'integer` » que doit avoir l'argument « `n` » afin de n'autoriser que des arguments dérivant de la classe `entiers`. Le type qui aurait été inféré par OCAML ne comprendrait qu'une contrainte sur la méthode `get_rep` que nous utilisons. Notons de plus que pour que le code soit correct il faut que les entiers que l'on utilise pour représenter un entier modulaire soient entre 0 et n. Dans la cas contraire la méthode `oppose` ne retourne pas un résultat correct.

Ce modèle possède donc plusieurs inconvénients :

- pour obtenir un niveau d'abstraction satisfaisant nous devons encore cacher la représentation effective.
- L'arité des fonctions n'est pas satisfaisante. Lorsque l'on dit à un mathématicien que 0 et « - » ont le même type, il est naturellement choqué. Il est en effet difficile de voir une constante et une fonction de la même manière.
- L'efficacité est problématique puisque tous les codes font en permanence de l'encapsulation/désencapsulation dans la variable d'instance `rep`.
- Pour la certification il est difficile de comprendre ce que doit faire une preuve sans formaliser complètement l'envoi de méthode.

5.1.4 Approche par représentation

Aucune des deux approches à base de modules ou à base de classes n'était donc réellement satisfaisante. On a bien vu que les opérations sont intrinsèquement liées aux ensembles. On a la fonction « $+$ » de $\mathbb{Z}$, la fonction « $+$ » de $\mathbb{R}$. Si on veut respecter à la fois la philosophie des langages à objets et les mathématiques on doit trouver un meilleur modèle.

L'idée que nous avons eu avec Sylvain Boulmé et Thérèse Hardin est de considérer que les ensembles doivent être des valeurs OCAML. Ainsi appeler la fonction « $+$ » de $\mathbb{Z}$ revient à envoyer la méthode `plus` à l'objet « `z` ». On obtient ainsi un double avantage, d'une part nous sommes plus proches des mathématiques. D'autre part on obtient un certain degré de surcharge (voir 4.3.2) puisqu'on peut distinguer `z#plus` et `r#plus` pour désigner respectivement les additions de $\mathbb{Z}$ et de $\mathbb{R}$ sous le même nom `plus`.

La question qui se pose est alors de savoir quels types doivent avoir les méthodes. En AXIOM c'est « `%` » qui joue ce rôle, c'est bien un type et il peut être utilisé pour décrire une signature. Un objet est une valeur et ne peut donc pas intervenir dans un type en OCAML. En effet les types et les valeurs sont dans des espaces séparés. L'idée que j'ai eue est d'expliciter la représentation que l'on utilise pour les éléments des ensembles. C'est en effet la représentation qui est effectivement manipulée par le programme. La représentation des données mathématiques est un des points fondamentaux du calcul formel et il est normal de lui donner un statut particulier.

J'ai donc été conduit à concevoir un modèle plus abstrait où les objets servent à donner une « vision mathématique » des valeurs qui sont manipulées. On peut ainsi dire que si on veut implémenter les entiers en utilisant les entiers d'OCAML le type `int` sert de *support* pour l'ensemble $\mathbb{Z}$. L'objet « `z` » donne alors une vue mathématique du type support. Les méthodes qu'on lui envoie manipulent directement des `int`. Ainsi la méthode `plus` de l'objet « `z` » aura le type OCAML

```
int -> int -> int
```

Nous sacrifions encore une fois l'abstraction de la représentation, mais nous avons gagné l'arité des fonctions. Nous sommes donc plus proches des mathématiques que dans le prototype naïf.

Si on veut pouvoir décrire des mathématiques il faut bien sûr que la représentation soit quelconque. Le programmeur doit en effet pouvoir utiliser n'importe quel support informatique pour implanter les éléments mathématiques qu'il manipule. De plus, on ne peut pas restreindre la liberté du programmeur qui doit pouvoir utiliser les structures de données qu'il souhaite pour implémenter ses algorithmes. La solution que nous proposons est donc d'ajouter un paramètre (qui sera la représentation des données) aux classes que nous écrivons. Ainsi pour décrire un monoide additif nous aurons :

```
class virtual ['rep] monoide_additif
object(the_mon)
  method virtual plus : 'rep -> 'rep -> 'rep
  method virtual zero : 'rep
end
```

On voit ici que l'ensemble que nous décrivons est `the_mon`. Il « porte » des signatures qui manipulent la représentation « `'rep` ». On garde les bonnes propriétés d'héritage et on peut construire des groupes additifs à partir de monoides additifs :

```
class virtual ['rep] groupe_additif
object(the_mon}
  inherits ['rep] monoide_additif
  method virtual oppose : 'rep -> 'rep
  method moins x y = the_mon#plus x (the_mon#oppose y)
end
```

Le type « `'rep -> 'rep -> 'rep` » de la méthode `moins` est automatiquement inféré par OCAML. On a la bonne arité des fonctions et le code qui définit la soustraction est beaucoup plus proche de ce que l'on écrirait en mathématiques :

$$x - y = x + (-y)$$

Si maintenant nous voulons implémenter les entiers à l'aide d'entiers OCAML, nous pouvons créer une classe

```
class entiers
  inherits [int] groupe_additif
  method zero = 0
  method plus x y = x+y
  method oppose x = -x
```

et produire l'objet « `z` » qui représentera $\mathbb{Z}$:

```
let z = new entiers
```

Nous voyons que les classes concrètes ne servent que de « générateurs » d'ensemble. L'état interne a disparu et les méthodes peuvent s'exécuter plus vite puisqu'il n'y a plus d'encapsulation comme dans le prototype naïf.

La difficulté est de programmer effectivement avec ce paradigme. Il faut en effet en permanence « déplier » la représentation pour donner les types des méthodes. On peut par contre bien paramétrer les structures mathématiques. Supposons que l'on ait une classe virtuelle paramétrée

```
['rep] corps
```

qui décrive les corps en représentation « `'rep` ». Un corps **K** est alors un objet « `k` » qui manipule des données de type « `'rep_k` ». Le type OCAML de l'objet « `k` » est bien sûr celui des objets qui donnent une vison mathématique de corps aux données de « `'rep_k` ». Soit

```
('rep_k) #corps
```

Un **K**-espace vectoriel **V** est un objet `vect` qui donne la vue mathématique d'espace vectoriel au type support « `'rep` ». Nous le décrivons dans une classe paramétrée par

- la représentation `'rep` des éléments de **V**
- la représentation `'rep_k` des éléments de **K**
- le type effectif « `'k` » de l'objet « `k` » qui représente le corps **K**

On a alors :

```
class virtual ['rep_k, 'k, 'rep] espace_vectoriel(k : 'k) =
object(vect)
  constraint 'k = ('rep_k) #corps
  inherits ['rep] groupe_additif
  method virtual multiplication_scalaire :
                'rep_k -> 'rep -> 'rep
end
```

La multiplication par un scalaire doit prendre en paramètres la représentation d'un élément λ de **K**, la représentation d'un élément v de **V** et retourner la représentation de l'élément λv de **V**.

Outre le paramètre de représentation nous avons du ajouter deux paramètres de type pour pouvoir paramétrer notre espace vectoriel **V** par le corps **K**. Plus généralement pour chaque « paramètre » que nous utilisons en mathématiques nous devrons écrire 2 paramètres de types dans la classe. C'est assez difficile à mettre en œuvre à la main mais on perçoit qu'une certaine automatisation est possible. En effet si on doit passer une collection C codée dans un objet « `c` » ayant une structure mathématique $\mathcal{S}$ codée dans une classe « `['rep] s` », le type de l'objet est un nouveau paramètre « `'c` » qui doit être contraint à avoir le type `('rep)#s`.

FIG. 5.1 – Les différents paradigmes

Foc	Axiom	Algèbre	Module	Objet Naïf	Objet Rep
entité	élément	élément	quelconque	objet	quelconque
opération	opération	loi, constante	composante	méthode	
collection	domaine	ensemble	module	classe	objet
signature				type de méthode	
espèce	catégorie	structure	type module	type classe	classe

On peut alors facilement implanter les entiers modulo n en paramétrant par une valeur « `n` » qui est la représentation utilisée par la classe `entiers` précédente.

```
class ['z] entiers_modulo (z:'z, n)
constraint 'z = #entiers
 method zero = z#zero
 method plus x y = let res = z#plus x y in
   res mod n
 method oppose x = z#moins n (z#oppose x)
```

Les types sont ici automatiquement vérifiés et inférés par OCAML. L'utilisation de la fonction `mod` d'OCAML est licite puisque la classe `entiers` utilise des `int` comme représentation.

L'utilisation que nous faisons des objets dans ce modèle est assez inhabituelle mais elle correspond bien à ce que l'on écrirait en mathématiques ou à ce que l'on implanterait dans un langage comme AXIOM. Le travail de thèse de Sylvain Boulmé a montré que ce modèle était cohérent. Il a pu complètement décrire en COQ cette utilisation des objets et dégager plus précisément les notions d'espèce et de collection.

On voit, en examinant le code, que les données qui sont manipulées dans ce prototype doivent être décrites par un type OCAML. Mais, pour décrire les polynômes récursifs en AXIOM, on utilise une véritable abstraction. On dit que les polynômes ont une variable « principale » et que les coefficients sont « récursivement vus » comme des polynômes en plusieurs variables. À notre connaissance seul le système de type d'AXIOM permettait de décrire cela de manière satisfaisante.

Les concepteurs d'AXIOM ont toujours pensé que la représentation devait être masquée à l'extérieur d'un domaine et que l'utilisateur ne devait en avoir qu'une vision abstraite. Le modèle que nous proposons doit lui complètement expliciter la représentation en exposant les types concrets. La différence est importante et il n'était pas évident que l'on puisse arriver à notre objectif des polynômes en représentation récursive.

En fait j'ai pu montrer que les données manipulées en calcul formel restent « simples ». Même dans le cas des polynômes en représentation récursive ce n'est pas la donnée qui est compliquée (c'est un simple type somme OCAML). C'est la vision « récursive» des coefficients comme polynômes en plusieurs variables qui est compliquée.

Supposons que l'on ait des polynômes en une variable en représentation creuse par exemple. Le support de la représentation creuse s'écrit facilement

```
type ('rep_a,'rep_n)univ_poly_rep = ('rep_a*'rep_n) list
```

si « `'rep_a` » et « `'rep_n` » sont les supports pour un anneau **A** et les entiers $\mathbb{N}$. C'est une simple liste de couples de $\mathbf{A} \times \mathbb{N}$.

Si on veut maintenant les polynômes récursifs à coefficients sur un anneau « de base » **B** utilisant un support « `'rep_b` », il nous suffit de substituer la représentation des polynômes récursifs à « `'rep_a` » dans le type précédent. On pourra alors définir le cas « inductif » où on interprétera la donnée comme un polynôme en sa variable principale :

```
type ('rep_b, 'rep_n) recursive_rep =
 | Base of 'rep_b
 | Inductive of
    (string *
      ((('rep_b, 'rep_n) recursive_rep)*'rep_n)univ_poly_rep)
```
(5.1)

Un élément est soit un élément de base de l'anneau **B** soit un élément ayant une variable principale (une `string` dans le code) et dont la valeur est la représentation creuse d'un polynôme en une variable à coefficients sur les polynômes récursifs. On a donc bien un type simple puisqu'écrit sans utiliser d'objets ni de modules. C'est simplement un type somme récursif. Nous reviendrons plus en détail sur les polynômes dans le chapitre 6, mais montrons ici que l'on peut exprimer les contraintes de types qui sont nécessaires dans notre modèle.

Supposons que l'on ait la classe abstraite `ring` pour modéliser un anneau **A** et la classe abstraite `entiers_positifs` pour modéliser $\mathbb{N}$. Supposons également que l'on ait la classe abstraite paramétrée

```
['rep_a, 'a, 'rep_n, 'n, 'rep]univ_poly(a :'a, n :'n)
```

pour modéliser des polynômes en une variable à coefficients sur **A**. Ici l'anneau **A** est codé par l'objet « `a` » de type « `'a` » qui doit être un anneau manipulant des données de type « `'rep_a` ». Les degrés sont dans $\mathbb{N}$, codé par l'objet « `n` » de type « `'n` ». On a donc dans la classe `univ_poly` les contraintes

```
constraint 'a = ('rep_a)#ring
constraint 'n = ('rep_n)#entiers_positifs
```
(5.2)

Puisque ses objets sont des anneaux on a aussi

```
inherits ['rep]ring
```
(5.3)

dans la classe `univ_poly`. Il nous faut une classe concrète qui effectue les opérations en représentation creuse. Supposons que cette classe soit :

```
['rep_a, 'a, 'rep_n, 'n]univ_poly_creux(a :'a, n :'n)
```

qui doit bien sûr hériter les opérations de la classe abstraite qu'elle implante. Sa définition contient donc

```
inherits
  ['rep_a, 'a, 'rep_n, 'n, ('rep_a, 'rep_n)univ_poly_rep]
   univ_poly
  (a, n)
```

Maintenant, pour écrire les polynômes récursifs à coefficients dans $\mathbf{A}$:

```
['rep_b, 'b, 'rep_n, 'n]recursive_pols(b :'b, n :'n)
object(rec_pols : 'rec_pols)
```
(5.4)

il nous faut dire que « `n` » modélise $\mathbb{N}$ et que « `b` » est un anneau :

```
constraint 'b = ('rep_b)#ring
constraint 'n = ('rep_n)#entiers_positifs
```
(5.5)

nous devons ensuite dire que c'est un anneau qui utilise la représentation récursive définie en (5.1). Nous le traduisons par un héritage

```
inherits [('rep_b, 'rep_n)recursive_rep] ring
```
(5.6)

Nous voulons pouvoir considérer le cas inductif du type `recursive_rep` de la définition (5.1) comme un polynôme en une variable muette, « étiqueté » par un nom de variable. Il nous faut donc un objet qui soit une instance de la représentation creuse à coefficients sur la représentation récursive. Nous pourrons alors appeler ses méthodes dans le corps de la classe. Il nous faut créer l'objet et le rendre accessible dans une méthode `up_rec` :

```
method up_rec = new(univ_poly_creux(rec_pols, n))
```
(5.7)

et le type de la méthode up_rec est automatiquement inféré par OCAML ! Nous voulons toutefois contraindre cette méthode afin de ne pouvoir sélectionner que les méthodes abstraites des polynômes en une variable. Ce qui nous conduit à forcer le type de up_rec à :

```
(('rep_a, 'rep_n)recursive_rep) as 'r_rep,
 'rec_pols,
 'rep_n,
 'n,
 ('r_rep*'rep_n)list,
)#univ_poly
```
(5.8)

Les deux premières lignes vérifient la contrainte d'anneau de (5.2), puisque d'après (5.4), l'objet rec_pols que nous définissons est bien de type

```
'rec_pols
```

C'est de plus un anneau qui utilise la représentation récursive à cause de (5.6). Les deux lignes suivantes sont les contraintes exprimées dans (5.5) et la dernière ligne est bien la substitution de (5.1).

5.1.5 Un bilan

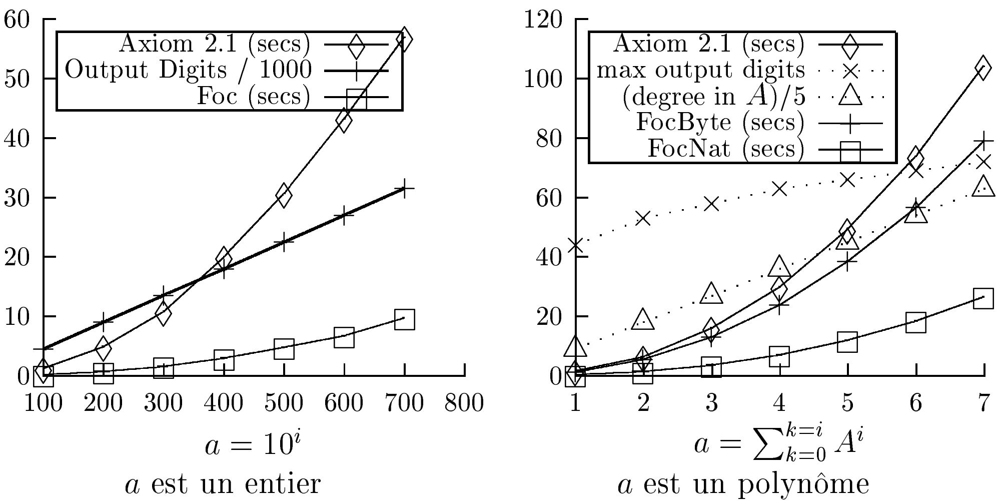
En 1999, nous commençons donc à bien comprendre ce que sont les espèces et les collections. Le modèle que nous avons dégagé est difficile à justifier à la main et le travail de Sylvain Boulmé montrant qu'il est cohérent était indispensable.

Grâce à la librairie OCAML nous savions que nous pouvions aller relativement loin dans l'implantation de mathématiques et d'algorithmes de calcul formel. Le code OCAML obtenu permettait d'implanter les polynômes récursifs en généralisant ce qui se faisait en AXIOM (voir la section 6.3). Nous avions pu implanter des représentations plus générales que celles d'AXIOM. Nous pouvions utiliser n'importe quelle librairie de grands nombres alors qu'AXIOM ne peut utiliser que les siens[1]. Pour les entiers modulaires nous pouvions utiliser des big_int OCAML, mais nous pouvions aussi utiliser des int. Ils sont plus petits et donc plus efficaces lorsqu'on travaille, comme c'est souvent le cas, modulo un petit entier.

En termes d'efficacité, le code obtenu était raisonnable puisqu'il permettait de calculer avec des tailles de polynômes qu'on ne pouvait pas atteindre

[1]En ALDOR on peut utiliser la librairie gmp, mais nous pouvions avoir plusieurs librairies dans le même code.

FIG. 5.2 – comparaison entre le prototype à base de représentation et AXIOM 2.2
Nous calculons ici le résultant des polynômes $P = x^{30} + ax^{20} + 2ax^{10} + 3a$ et $Q = x^{25} + 4(a+1)x^{15} + 5(a+1)x^5$.



Les représentations sont analogues en AXIOM et en FOC. C'est le même algorithme de résultant qui est codé dans les deux cas.

en AXIOM. La figure 5.2 montre qu'AXIOM se comporte mal quand les entiers sont gros. Pour des tailles d'entiers d'une cinquantaine de chiffres, AXIOM se comporte comme du « byte code » OCAML qui est beaucoup moins rapide que le code natif produit par OCAML.

Pour la sûreté d'exécution nous avions obtenu un code dans lequel des contraintes provenant des mathématiques étaient insérées. Nous devions expliciter l'ensemble de la donnée mais la cohérence globale était vérifiée par le compilateur OCAML. En AXIOM, la donnée est immédiatement masquée mais la construction `pretend` permet au programmeur de « forcer le type » à ce qu'il veut (c'est le `Obj.magic` d'OCAML). Elle est simplement utilisée par le compilateur pour masquer le type concret.

Les travaux du groupe ATYPICAL (voir [Tho]) de l'Université de Kent, sur l'ajout de composantes logiques à ALDOR, nécessitent d'éliminer la construction `pretend` du langage. Le projet FoC collabore avec ATYPICAL à travers le projet ALLIANCE d'échanges franco anglais. Comme nous, ils ont été amenés à expliciter la représentation et à garantir une cohérence globale pour pouvoir supprimer cette construction.

C'est finalement ce paradigme d'objets que nous avons retenu pour le projet. Il est toutefois impossible de demander à un programmeur mathématicien de dériver les contraintes de (5.8). Il faut pour cela une connaissance des langages à objets et de leur sémantique qu'il n'a pas.

En nous appuyant sur la réalisation de prototypes on avait pu dégager quelques mots clefs, « espèce », « collection », « hérite » ... Nous avions l'embryon d'une syntaxe concrète et une « idée » de sa signification. Le travail de thèse Sylvain Boulmé (voir [Bou00]), encadré par Thérèse Hardin et moi même, a donné un sens à tous ces mots clefs pour en avoir une spécification formelle. Il fallait encore la mettre en œuvre pour en dégager un véritable langage, avec une analyse du code ... C'est ce qui constitue le travail de thèse de Virgile Prévosto (voir [Pre00, Pre01]) encadré par Thérèse Hardin et Damien Doligez. La thèse de Stéphane Fechter (voir [Fechter01]), encadrée par Thérèse Hardin et Catherine Dubois, vise elle à donner une sémantique à FoC.

5.2 Notations

Nous noterons les fonctions comme en OCAML, c'est à dire que si A, B et C sont trois ensembles nous noterons les fonctions de $A \times B \to C$ de manière curryfiée $A \to B \to C$. En AXIOM on noterait $(A, B) \to C$.

Les types seront notés à la manière de COQ, ainsi les listes dont les termes sont dans A seront notées $\texttt{list}(A)$. La dénomination que nous prendrons pour les noms de fonctions comportant plusieurs mots sera inspirée d'OCAML et nous séparerons les mots avec le caractère souligné comme dans `une_function`. En AXIOM on séparerait les mots en mettant une majuscule aux mots après le premier comme dans `uneFoncion`.

Nous définirons une opération par le mot clef `let`, nous en déclarerons l'existence par le mot clef `sig`. Ces notations sont directement inspirées d'OCAML. En AXIOM c'est « `==` » qui permet de définir une opération.

Nous séparerons différentes définitions par le mot clef `and` et la définition sera (globalement) récursive par ajout du mot clef `rec` après le `let`. En AXIOM les définitions sont toujours récursives sans qu'il soit besoin de le préciser. Par exemple dans les définitions AXIOM

```
fib(n) == n
```
(5.9)

```
fib(n) ==
  n=0 => 1
  n=1 => 1
  fib(n-1)+fib(n-2)
```
(5.10)

les occurrences de `fib` dans le corps de la fonction de (5.10) désignent l'opération que l'on est en train de définir, alors qu'en FOC elles désigneraient l'identificateur `fib` défini *avant* (en (5.9) donc). Ce choix est compatible avec ce qui est fait en OCAML où on analyse complètement l'expression avant de lui donner un nom.

Nous suivrons la convention traditionnelle des langages à objets en notant `self` l'objet en cours de définition. En AXIOM c'est le mot clef « `%` » qui joue ce rôle.

En ce qui concerne les identificateurs, nous prendrons la convention qui consiste à préfixer ces identificateurs par « `!` » ou « `#` ». Plus précisément nous utiliserons « `!` » lorsque l'identificateur désigne une valeur gérée par le mécanisme de FOC. Il s'agit alors d'une méthode d'un objet. En AXIOM cette convention n'existe pas puisque la surcharge (voir 4.3) est autorisée, on peut toutefois désambiguïser une opération à l'aide de « `$` ». Ainsi en AXIOM on pourrait noter « `op$%` » ce qui se noterait `self!op` avec nos conventions. Nous préfixerons par « `#` » lorsqu'il s'agit d'un identificateur global, c'est à dire un identificateur défini en dehors du mécanisme de FOC. Un identificateur non préfixé est un paramètre introduit par l'utilisateur.

Un mathématicien est amené à considérer des éléments de certains ensembles pour décrire certaines opérations. Il est aussi amené à faire une théorie générale pour des ensembles ayant une structure semblable et doit nommer un ensemble particulier ayant cette structure. On notera « `x in a` » pour désigner $x \in a$, on notera « `a is ring` » pour dire que a est un anneau. Nous respectons ici l'usage FoC qui fait la différence entre les éléments d'un ensemble et la structure mathématique que doit avoir cet ensemble. En Axiom on noterait les deux de la même façon « `x : a` » et « `a : ring` ».

5.3 Types et valeurs

Les *entités* FoC sont les éléments mathématiques qui sont manipulés par les programmes et modélisés par le programmeur. Elles ont un aspect abstrait d'« éléments » d'un type abstrait de données mais elles ont aussi un aspect concret qui est la *représentation* que le programmeur souhaite leur donner. Cette représentation est le type de données qui est reflété dans le langage d'exécution OCaml.

FoC possède donc un système de types simples qui sont les types que l'on peut définir dans un langage de programmation fonctionnel d'usage général.

5.3.1 Types atomiques

FoC possède des types « atomiques » comme `bool`, `int` ... À chaque type atomique sont associées des opérations. Ce sont les types qui sont directement importés du langage de programmation (OCaml donc) sous-jacent à FoC.

5.3.2 Constructions

FoC permet de définir des types construits. Si τ_1 et τ_2 sont des types alors

– les fonctions de τ_1 dans τ_2 ont un type noté τ_1`->`τ_2. Les valeurs fonctionnelles sont introduites par la construction `fun` en notation curryfiée. Ainsi

$$\texttt{fun } x \texttt{ -> fun } y \texttt{ -> } \mathcal{E}(x, y)$$

désigne une fonction a deux arguments.

– l'union disjointe d'éléments de τ_1 et de τ_2 a un type. On doit nommer les différentes composantes d'une union. Par exemple

```
type mes_listes =
  | Vide in mes_listes
  | Nonvide in int -> mes_listes-> mes_listes
```

La notation vient ici de Coq et apparaît plus généralement comme une suite d'assertions dont on doit nommer les branches. Les noms que l'on donne à ces branches apparaissent comme les injections canoniques de

chacun des ensembles dans l'union disjointe. Dans l'exemple précédent tout se passe comme si `Vide` était une constante de `mes_listes` et `Nonvide` une fonction à deux arguments. La notation est consistante avec celles de la section 5.2.
La programmation avec des types somme se fait *par filtrage* ce qui correspond à une définition par cas en mathématiques. Ainsi

$$\begin{array}{l}\texttt{let rec foo}(l) = \texttt{match}\, l\, \texttt{with} \\ \quad \texttt{| \#Vide} \rightarrow 0 \\ \quad \texttt{| \#Nonvide}(h,t) \rightarrow h + \texttt{foo}(t)\end{array}$$

permet d'évaluer « $h + \texttt{foo}(t)$ » dans un contexte où h, t et l sont des identificateurs liés aux bonnes valeurs. Ce mécanisme permet d'introduire de nouveaux noms sans faire explicitement appel à une fonction qui déstructure la donnée. En AXIOM il faudrait explicitement tester et déstructurer la donnée à l'aide de `case` et d'une coercion (notée « :: »).

$$\begin{array}{l}\texttt{foo}(l) == \\ \quad l\; \texttt{case Vide} \rightarrow \\ \quad\quad 0 \\ \quad l\; \texttt{case Nonvide} \rightarrow \\ \quad\quad (h,t) == l :: \texttt{Nonvide} \\ \quad\quad h + \texttt{foo}(t)\end{array}$$

5.3.3 Paramètres

On peut paramétrer des types par d'autres types en utilisant des identificateurs de types notés comme en OCAML « `'a` », « `'b`» ... On peut alors écrire n'importe quelle expression de type formée des constructions précédentes en considérant que les identificateurs de types sont des types atomiques. Un type paramétré est alors formé d'un nom, d'identificateurs de types et d'une expression de type ne contenant pas d'identificateurs de types libres. Par exemple les listes pourraient s'écrire de la manière suivante

```
type listes('a) =
  | Vide in listes('a)
  | NonVide in 'a -> listes('a) -> listes('a)
```

Notons que le système de types ainsi défini permet l'inférence de types. Si une expression est typable a un type unique et on peut déterminer à la fois l'existence d'un type et le type qu'elle a. On utilise pour cela un algorithme d'inférence de types analogue à celui de CAML-LIGHT.

5.3.4 Polymorphisme

Pendant l'inférence de types on est amené à introduire des variables de types qui peuvent être généralisées. On a alors un système de types polymorphes. Ainsi pour typer l'expression

```
fun x -> x
```

on est amené à effectuer le typage du corps de la fonction dans un environnement où `x` a un type désigné par une inconnue de type τ_x à déterminer. On trouve donc l'expression de type $\tau_x \to \tau_x$ comme type pour la fonction. L'inconnue devient une variable de type qui se trouve libre. On peut généraliser cette expression de type en $\forall\tau\, \tau \to \tau$ qui est un type polymorphe.

5.3.5 Comparaison avec Axiom

Le système de types de FoC diffère donc de celui d'Axiom. Les types de FoC modélisent les données au sens informatique alors que les types d'Axiom visent à modéliser à la fois les mathématiques et les données dans le même formalisme.

En particulier nous n'autorisons pas la définition d'un type FoC avec des paramètres de valeurs FoC. C'est le mécanisme de *collections* et d'*espèces* de FoC qui va permettre de donner aux valeurs d'un type FoC une interprétation mathématique en termes d'ensembles et de structures algébriques.

Ceci permet de séparer complètement le typage des expressions FoC de leur évaluation puisque l'espace des types et celui des valeurs sont ainsi complètement distincts. L'algorithme de typage est ainsi indépendant de l'algorithme d'évaluation et peut être fait statiquement. En Axiom par contre l'algorithme de typage doit faire appel à celui d'évaluation ce qui pose des problèmes de cohérence et de terminaisons décrits, par exemple, dans [PT00].

Une conséquence est qu'en FoC nous pouvons directement avoir des opérations polymorphes. En Axiom le polymorphisme est plus compliqué à décrire car l'utilisateur doit lui même expliciter les types. Prenons l'exemple très simple de la fonctionnelle `map` que l'on peut définir sur le types `listes` précédent en dehors du mécanisme FoC

```
let rec map(f,l) = match l with
  | #Vide() -> #Vide()
  | #NonVide(fst,rst) -> #NonVide(f(fst),#map(f,rst))
```

La définition ressemble ici beaucoup à ce que l'on écrirait en OCaml et le type inféré est bien $\forall\alpha,\beta\ (\alpha \to \beta) \to \mathtt{listes}(\alpha) \to \mathtt{listes}(\beta)$ et l'on peut instancier α et β avec n'importe quels types.

En AXIOM par contre c'est à l'utilisateur d'écrire ce code dans un « package » et d'expliciter les paramètres nécessaires :

```
ListMappingPackage(A:BasicType, B:BasicType): PUB == PRIV where
  PUB ==
    map : (List(A), (A->B)) -> List(B)
  PRIV ==
    map(f,l) ==
     if empty?(l)
     then l
     else cons(f(first(l)), map(f, rest(l))
```

Ensuite, lorsque l'utilisateur veut se servir de sa fonctionnelle `map`, il doit dans son code source, expliciter quelle instance il veut en utiliser. Par exemple en ALDOR si D_1 et D_2 sont des domaines et que l'on veut utiliser `map` avec $f : D_1 \to D_2$ et $g : D_2 \to D_1$ on doit écrire

```
import from ListMappingPackage(D1,D2)
import from ListMappingPackage(D2,D1)
```

Ceci donne un travail important de typage, conception et paramétrisation au programmeur mathématicien qui n'y est pas formé. Sur des exemples plus compliqués, un programmeur non averti peut facilement se tromper dans l'instanciation des paramètres. Le concepteur de la fonctionnelle `map` doit lui aussi paramétrer son package de la manière la plus générale possible. Ceci n'est pas toujours facile et, par exemple, dans certaines versions d'AXIOM l'affichage de listes sur un domaine D n'est défini que dans le cas où D est un ensemble (`SetCategory` au sens AXIOM). Cette opération est pourtant parfaitement définie sur n'importe quel domaine ayant une opération d'impression (provenant donc de `CoercibleTo`(`OutputForm`) au sens AXIOM).

5.4 Espèces et Collections

Une fois les types et les valeurs définis nous devons en donner une vision plus proche de la pratique des mathématiques et du calcul formel. Nous avons maintenant des entités et un type support qui est leur représentation. Il nous faut abstraire ce type dans des ensembles possédant une structure algébrique comme on fait habituellement en mathématiques.

Les *collections* FOC visent à modéliser la notion d'ensemble en mathématiques. Elles regroupent une famille d'entités cohérentes, c'est à dire ayant même *représentation*, pour en donner une vision plus proche des mathématiques. Leur rôle est d'encapsuler les opérations disponibles sur les entités qu'elle « contiennent » et de masquer leur représentation effective en un type abstrait.

Les *espèces* FoC visent à modéliser la notion de structure algébrique en mathématiques. Une espèce est formée d'un ensemble de champs qui sont ses *composantes*. Elles spécifient ou implantent les opérations qui sont disponibles sur les entités d'une collection.

5.4.1 Les composantes

Une composante est nécessairement nommée par le programmeur et on peut l'assimiler au nom qu'il lui a donné. La première composante d'une espèce est sa représentation, donnée par le mot clef `rep`. Cette représentation s'exprime dans le langage de types de FoC et peut être implicite ou explicite.

Une opération peut être simplement déclarée par le mot clef `sig`. On doit donner son type dans le langage de type de FoC en considérant que le mot clef `self` désigne un type. Par exemple dans

```
species monoide_additif
  sig plus in self -> self -> self
  sig zero in self
end
```

on spécifie d'abord une opération binaire puis un élément nommé `zero`. Lors de l'instanciation de l'espèce c'est le type support qui est effectivement utilisé.

Une espèce peut étendre les composantes d'une autre espèce et en définir de nouvelles. Une composante peut être implantée par le mot clef `let` c'est à dire qu'une implémentation en est donnée. Ainsi dans

```
species groupe_additif
  inherits monoide_additif =
  sig oppose in self -> self;
  let moins(x,y) = self!plus(x,self!oppose(y));
end
```

on implante une opération binaire de soustraction dont le type FoC est

```
self -> self -> self
```

ce qui est inféré par le compilateur FoC. En effet dans un environnement où `self` est un type et où les types de `plus` et de `oppose` sont connus, celui de `moins` s'en déduit.

Certaines opérations se définissent de manière récursive et on doit préciser si une déclaration est récursive par le mot clef `let rec`. Ceci permet de donner un type au corps d'une définition.

Les propriétés des opérations d'une espèce peuvent également être spécifiées par le mot clef `property` ou démontrées par le mot clef `theorem`. Nous n'en parlerons pas ici.

Une espèce peut implanter un nombre arbitraire de composantes. Par exemple, si on veut obtenir les entiers, on peut spécifier un type support et implémenter les opérations `oppose`, `plus` et `zero` dans une espèce.

```
species mes_entiers
  inherits groupe_additif =
    rep = int ;
    let oppose(x) = -x ;
    let plus(x,y) = x+y ;
    let zero = 0;
end
```

Une telle espèce est dite *concrète* car elle implante toutes ses composantes. On peut toutefois redéfinir certaines de ces opérations. Elle n'est donc pas figée.

Dans l'exemple précédent c'est le type `int` (donné par `rep`) qui est substitué à `self` et la signature `self -> self -> self` devient `int -> int -> int`. Ces types concrets sont masqués à l'extérieur de l'espèce en faisant l'assimilation entre `self` et le type effectivement donné dans la représentation.

Donc dans l'espèce `mes_entiers` les types mis à la disposition du programmeur sont :

```
species mes_entiers =
  sig plus in self -> self -> self
  sig zero in self
  sig oppose in self -> self
  sig moins in self -> self -> self
end
```

Bien sûr le compilateur vérifie le typage mais il ne fait pas automatiquement le lien entre le type donné par `rep` et `self`. Comme en AXIOM où les opérations sont déclarées, le programmeur doit préciser s'il s'agit du type concret `rep` ou bien de sa vision abstraite `self`.

Ainsi pour étendre l'espèce `mes_entiers` avec une méthode `multiplie` si l'on écrit :

```
species entiers_bis
  inherits mes_entiers =
  let multiplie(x,y) = x*y
end
```

le type de `multiplie` n'est pas explicité et l'espèce sera vue avec un type

```
int -> int -> int
```

pour `multiplie`. Par contre, on aura

```
self-> self -> self
```

si on écrit :

```
species entiers_bis
  inherits mes_entiers =
  let multiplie(x in self, y in self) in self = x*y
end
```

il y a inférence de types et vérification que le type inféré est bien celui donné par `rep`. On fait alors la correspondance entre la donnée concrète de `rep` et sa vue abstraite dans `self`.

On peut donc voir le mécanisme des espèces comme un moyen d'aller du général au particulier ou encore d'une théorie purement abstraite à un modèle entièrement explicite de cette théorie.

5.4.2 Le gel

Lorsque toutes les composantes d'une espèce sont implantées on peut vouloir l'utiliser dans des programmes. Il faut alors la « geler » pour en faire une *collection* qui apparaît donc comme la réalisation d'une espèce :

```
collection z implements mes_entiers =
end
```

Cette réalisation joue le même rôle que la création de nouveaux objets d'une classe. Une collection a toutefois le droit de redéfinir n'importe quelle opération de l'espèce qu'elle implante. Par exemple, dans :

```
collection z_bis implements mes_entiers =
  let moins(x,y) = x-y ;
end
```

tout se passe comme si on créait une espèce avec les redéfinitions avant de créer effectivement une collection. Une collection est donc de nature statique (voir la section 4.3.1) et ne peut plus être modifiée.

5.4.3 Paramétrage

On est parfois amené à considérer d'autres ensembles que celui que l'on est en train de décrire. Il nous faut pouvoir écrire des signatures qui les font intervenir. On peut donc avoir des paramètres dans la définition d'une espèce. Dans

```
species espace_vectoriel(k is corps)
  inherits groupe_additif =
  sig multiplication_scalaire in k -> self -> self;
  let oppose(x) = multiplication_scalaire(k!oppose(k!un), x);
end
```

on spécifie une opération binaire dont le premier argument est dans le corps qui est passé en paramètre de l'espèce. On exprime les signatures en étendant le langage de types de FoC. On autorise les paramètres de l'espèce qui sont des collections à être vus comme des paramètres de type (en plus de `self` bien sûr). Lors de l'instanciation de l'espèce le type qui sera réellement utilisé par l'opération sera $\tau_{\mathrm{k}} \rightarrow \tau_{\mathrm{self}} \rightarrow \tau_{\mathrm{self}}$ où τ_{k} désigne le type FoC qui est la représentation effective de la collection « `k` » passée en paramètre à l'espèce `espace_vectoriel`. Le type τ_{self} est le type spécifié par `rep` lorsqu'il est complètement instancié.

Nous avons défini l'opération `oppose` qui était simplement déclarée dans l'espèce `groupe_additif`. Le type inféré par FoC pour cette composante est

```
self -> self
```

qui est bien le même type que celui déclaré dans `groupe_additif` et la définition est cohérente.

Une espèce peut également être paramétrée par des entités et on doit préciser à quelle collection elles appartiennent. Ainsi en reprenant la collection « `z` » on peut écrire l'espèce des entiers modulo n :

```
species entiers_modulo(n in z)
  inherits ring =
  ...
end
```

En mathématiques ou en calcul formel nous sommes souvent amenés à décrire des dépendances plus compliquées. Dans l'exemple précédent la collection « `z` » est entièrement statique et on veut parfois passer en paramètres à la fois une entité et la collection qui le contient.

Par exemple si $\mathbf{R}$ est un anneau euclidien et si r est un élément de $\mathbf{R}$ on veut parfois travailler dans l'anneau quotient $\mathbf{R}/<r>$ où $<r>$ est l'ensemble des multiples de r. Un élément de $\mathbf{R}/<r>$ est une classe d'équivalence de la relation définie par : deux éléments x et y sont équivalents si et seulement si r divise leur différence. Les éléments de $\mathbf{R}/<r>$ peuvent alors être représentés par des éléments de $\mathbf{R}$ qui sont réduits modulo r. Ainsi en FoC on peut écrire

```
species modular_ring(a_ring is ring, an_elt in a_ring)
  inherits ring =
  ...
```

On a ici une dépendance plus compliquée dans la mesure où le « type » de `an_elt` est `a_ring` qui est la « valeur » du premier argument. Le type de `an_elt` est en fait celui de la représentation des entités de `a_ring`. Celui de `a_ring` est celui des objets de la classe `a_ring_class` implantant `a_ring`. Au cours de la compilation de FoC vers OCaml on ajoute donc des contraintes de types comme dans la section 5.1.4.

5.4.4 L'héritage

Nous avons vu dans la section 4.3 que les langages à base d'objets offraient des mécanismes d'héritage différents. Des langages comme Java offrent une notion d'héritage simple qui est insuffisante pour décrire les mathématiques en conservant une structure proche de l'intuition. Dans le chapitre 5 nous avons naturellement implanté une $\mathbf{K}$-algèbre avec de l'héritage multiple. Les espèces FoC peuvent donc hériter de plusieurs autres espèces. Par exemple, pour définir une algèbre en FoC nous écrivons simplement :

```
species algebre(k is field)
  inherits anneau,
           espace_vectoriel(k) =
  sig lift in k -> self;
  let multiplication_scalaire(a,x) = !multiplie(!lift(a),x);
  end
```

Ici, `lift`, est le nom que l'on donne au morphisme d'anneau de $\mathbf{K}$ dans la $\mathbf{K}$-algèbre $\mathbf{A}$.

On est bien ici en présence d'héritage multiple. Les espèces `anneau` et `espace_vectoriel` partagent au moins les opérations de groupe additif dont certaines sont implantées dans l'espèce `espace_vectoriel`.

Lorsqu'une composante est héritée plusieurs fois, on doit s'assurer que toutes ses instances sont compatibles entre elles. En FoC, elles doivent avoir le même type. C'est alors la dernière définition dans l'ordre d'héritage qui est choisie. Ces règles sont les mêmes que celles d'OCAML et l'héritage des espèces peut se voir comme l'héritage des classes en OCAML.

Il est à noter que la représentation des entités est une composante de l'espèce, elle est héritée comme les autres composantes. Ainsi l'espèce `entiers` plus haut est encore dérivable, les contraintes de compatibilité de types font toutefois qu'une redéfinition de `rep` en un autre type que `int` provoque une erreur.

5.4.5 Surcharge

Comme le modèle sous-jacent est celui à base de représentation, les méthodes sont envoyées aux objets et donc aux collections. Ainsi une méthode définie dans une espèce $\mathcal{E}$ peut être utilisée sous le même nom par plusieurs collections dérivant de $\mathcal{E}$. Par exemple l'addition spécifiée dans l'espèce `monoide_additif` pourra être envoyée à la collection « `z` » ou alors à la collection `z_2_z`

```
let deux = z!plus(z!un,z!un) ;;
collection z_2_z implements entiers_modulo(#deux);;
```

On aura ainsi deux opérations `z !plus` et `z_2_z !plus`. Nous obtenons ainsi un certain niveau de surcharge. Par contre, dans l'espèce `algebre` nous n'autorisons pas la surcharge pour la multiplication. On doit avoir deux noms de méthodes distincts :

```
sig mult : self -> self -> self
sig multiplication_scalaire : k -> self -> self
```

AXIOM, en revanche, caractérise une opération par son nom et son type. Le nom peut être surchargé puisque c'est le couple qui caractérise l'opération. Les deux opérations ci dessus peuvent donc s'appeler `mult`.

5.4.6 Comparaison avec Axiom

En première approximation, une espèce FoC ressemble à une catégorie AXIOM et une collection FoC ressemble à un domaine AXIOM. Notons toutefois un certain nombre de différences.

L'héritage des catégories en AXIOM (la construction `Join`) ressemble à celui de FoC, mais nous n'avons pas réussi à savoir dans quelle mesure ils sont compatibles. La question est rendue d'autant plus difficile qu'AXIOM autorise un mécanisme d'héritage au niveau des domaines grâce à la construction `add` qui permet d'ajouter ou de redéfinir n'importe quelle opération d'un domaine. On a donc en AXIOM un double mécanisme où un domaine réalise une (ou plusieurs) catégorie et où il ajoute des opérations à un autre domaine.

La liaison tardive de FOC (voir la section 4.3.5) semble ne pas exister en AXIOM. Les définitions « par défaut » dans les catégories sont bien prises en compte si aucune autre définition n'existe. On a bien un comportement « retardé ». Mais, dans un domaine, la liaison ne semble pas être « retardée ». La portée d'un identificateur ne s'étend jamais aux domaines qui ajoutent des opérations au domaine en cours. Ainsi l'exemple `pair` et `impair` de la section 4.3.5 se comporte statiquement. Il semble donc qu'on a bien une notion de redéfinition mais pas de liaison tardive.

Dans une approche à base d'objets, il nous à semblé plus cohérent et plus agréable pour le programmeur, d'avoir un comportement uniforme. La liaison tardive permet souvent d'écrire moins de code que la liaison statique. Par expérience, en AXIOM, j'ai souvent dû réécrire tout le corps d'un domaine pour changer le comportement de quelques fonctions qui sont cruciales pour l'efficacité de certains algorithmes.

ALDOR autorise l'« extension » d'un domaine grâce à la construction `extend`. Je n'ai jamais bien compris cette construction, elle est censée permettre d'ajouter des opérations à un domaine sans le renommer. Par exemple on peut avoir une vision très simple des entiers (`Integer`), avec simplement les fonctions successeur et prédécesseur. De tels `Integer` peuvent être utilisés dans des signatures. Lors des développements, on souhaite enrichir ces `Integer` et en faire, par exemple, un anneau euclidien. Les signatures existantes faisant intervenir `Integer` ne doivent par contre pas changer.

Appelons I_1 les `Integer` avant extension et I_2 ceux après extension. Il se peut parfaitement que I_2 redéfinisse des fonctions de I_1 à l'aide de nouvelles opérations. Prenons une opération de nom « `op` », O_1 sa définition dans I_1 et O_2 sa redéfinition dans I_2. Si O_2 appelle une opération de nom `aux` qui n'existe pas dans I_1, on ne peut l'appeler que dans un contexte où le nom `aux` est défini. C'est donc la définition O_1 de `op` qui doit être vue puisque `aux` ne peut pas être appelée dans un contexte où elle n'existe pas. Les nouveaux `Integer` I_2 ne peuvent donc pas masquer les anciens `Integer` I_1.

En FOC la portée des collections est statique et on doit définir une *nouvelle* collection. On peut bien sûr lui donner le même nom mais on a deux collections C_1 et C_2 portant le nom « `c` ». Nous sommes sûrs que le code qui utilisait sous le nom « `c` » les algorithmes de C_1 n'utilise pas une redéfinition de C_2.

La construction `extend` facilite l'ecriture de la librairie mais, son manque de clarté fait que nous n'avons pas voulu introduire ces problèmes dans nos prototypes de librairies. On a donc choisi de ne pas utiliser des collections explicitement créées dans les signatures de la librairie.

Ce choix me semble cohérent avec la portée des noms dans un langage de programmation. Lorsqu'un programmeur renomme une valeur c'est qu'il n'en a plus besoin, puisqu'il s'est enlevé la possibilité de l'utiliser. C'est le « run-time » qui décide de son utilité dans d'autres contextes pour pouvoir, éventuellement, récupérer la mémoire qu'elle occupe.

Chapitre 6

Implémentation des polynômes en FoC

La définition des polynômes donnée dans la section 1.1 n'est pas constructive. Elle ne donne pas de manière de « représenter » un polynôme ni de moyen d'« effectuer » les opérations sur ce codage.

Du point de vue du calcul effectif sur les polynômes de $\mathbf{A}[X]$, le calcul formel distingue deux grands types de représentations, celles dites creuses à base de listes et celles dites denses à base de vecteurs.

En algorithmique, le modèle des listes suppose l'accès en temps constant au premier élément de la liste ainsi qu'à la liste de ses successeurs, c'est le cas quand on définit un type somme en FoC comme dans la section 5.3. Celui des vecteurs suppose l'accès en temps constant à n'importe quel élément du vecteur. Ce modèle ne correspond à la réalité du calcul formel que dans les cas, très particuliers, où les coefficients des vecteurs sont de taille fixe et connue à l'avance, mais c'est rarement le cas en calcul formel. Même si on peut borner la taille des coefficients, il est souvent utile de ne pas allouer la taille maximum. Ces représentations sont par contre bien adaptées aux calculs modulaires où la taille des coefficients ne grossit pas pendant les calculs. Comme nous aurons, en général, des tailles de coefficients variables, nous privilégions les représentations creuses.

On code souvent un monôme par un couple formé de son coefficient et de son monôme primitif comme dans la section 1.1. Dans les représentations creuses, on ne code que les monômes dont le coefficient est non nul. Même dans ce cas, nous avons besoin d'un ordre bien fondé sur les monômes primitifs des monômes du polynôme, afin de garantir la terminaison de beaucoup d'algorithmes.

Pur pouvoir multiplier facilement un polynôme par un monôme primitif sans réordonner les monômes du résultat, on choisit un ordre compatible avec la multiplication des monômes. C'est à dire que le monoide multiplicatif des monômes primitifs est ordonné avec un ordre compatible avec la multiplication. Pour les polynômes en une variable, l'ordre sur les monômes primitifs $\{X^i\}_{i\in\mathbb{N}}$ est donné par l'ordre sur les entiers et la multiplication des monômes primitifs correspond à l'addition des entiers.

Pour des raisons évidentes d'accès au degré et au coefficient dominant d'un polynôme en temps constant, on décide souvent de représenter les polynômes en une variable par une liste de monômes triée dans l'ordre décroissant des monômes primitifs qui y apparaissent.

En reprenant les définitions de Lang (voir [Lan69]), on forme les polynômes en plusieurs variables en se donnant un ensemble $\mathcal{S}$ de « variables » et en formant le monoide multiplicatif $\mathcal{M}(\mathcal{S})$ des fonctions de $\mathcal{S}$ dans $\mathbb{N}$ qui sont nulles presque partout[1]. On définit la multiplication de deux éléments m_1 et m_2 dans $\mathcal{M}(\mathcal{S})$ par $(m_1m_2)(s) = m_1(s) + m_2(s)$, ce qui forme le monoide commutatif libre noté multiplicativement. On peut ensuite former l'algèbre libre comme dans la section 1.1, pour obtenir $\mathbf{A}[\mathcal{S}]$ les polynômes dont les variables sont dans $\mathcal{S}$.

La librairie FoC implante la notion de monoide multiplicatif, muni d'un ordre, bien fondé, compatible avec la multiplication. Ces opérations sont spécifiées dans l'espèce `ordre_monomial`. Elles sont illustrée dans la figure 6.3 page 108. On décrit ainsi toutes les opérations sur les monômes primitifs. Nous avons choisi de ne pas représenter directement le monoide multiplicatif mais l'ensemble des degrés et donc d'avoir une notation additive. Ainsi $X^iX^j = X^{i+j}$ même si $i = i_1 \ldots i_n$ et $j = j_1, \ldots j_n$ puisqu'on peut interpréter X comme l'ensemble $X_1 \ldots X_n$.

Les polynômes en une variable en sont une simple spécialisation. On ainsi un meilleur niveau de réutilisation du code que dans les librairies AXIOM ou ALDOR que nous connaissons.

6.1 Représentations distribuées

D'un point de vue calculatoire, le calcul formel distingue deux grands types de représentations pour les polynômes en plusieurs variables, celles dites distribuées et celles dites récursives. En général, on considère que les représentations distribuées sont « adaptées» aux méthodes globales de résolution d'équations qui gèrent, en bloc, toutes les variable d'un système. Souvent, les représentations récursives sont mieux « adaptées » aux méthodes qui privilégient une variable « principale » dans la résolution.

[1] c'est à dire nulle sauf en nombre fini de points, comme dans la section 1.1

Dans les représentations distribuées, on représente les polynômes de la même façon que les polynômes en une variable mais en prenant un monoide plus général. On se contente souvent d'un nombre fini de variables, et on prend le monoide commutatif libre engendré par ces variables. On a encore une fois le problème d'obtenir un ordre bien fondé sur les monômes primitifs qui interviennent dans un polynôme. Nous avons pour FoC repris ce schéma en faisant une seule implémentation des polynômes distribués, paramétrée par un ensemble de monômes primitifs munis d'un ordre monomial comme décrit précédemment.

Les polynômes en plusieurs variables sont couramment utilisés en calcul formel pour la résolution d'équations et notamment dans les bases de Gröbner où les ordres sont extrêmement importants. On se donne donc un ensemble fini $\{X_1, \ldots X_n\}$ de variables et on identifie le monôme $X_1^{i_1}, \ldots X_n^{i_n}$ avec le n-uplet $(i_1, \ldots i_n)$. Traditionnellement, les ordres les plus utilisés en calcul formel sont :

– l'ordre lexicographique direct,
– l'ordre lexicographique inverse.

Ces ordres sont toutefois mal adaptés à certains calculs. On leur préfère, en général, des ordres basés sur le degré total (la somme des degrés en chaque variable) d'un monôme primitif :

– l'ordre du degré raffiné par l'ordre lexicographique direct
– l'ordre du degré raffiné par l'ordre lexicographique inverse.

Ils sont tous implantés en FoC en utilisant la possibilité de paramétrer par des entités quelconques en FoC et donc par des fonctions.

Nous pouvons toutefois remarquer, d'un point de vue de spécifieur, que les premiers sont plus « généraux ». Limitons nous au cas de deux variables, pour « définir » l'ordre lexicographique, il suffit d'avoir un ordre bien fondé sur chacune des variables. On peut donc imaginer des représentations différentes, voire des « paquets » différents de variables. Par contre, lorsque nous faisons la somme des degrés en chaque variable, nous sommes forcés de considérer que les degrés sont dans le même ensemble.

6.2 Représentations récursives

Les représentations récursives privilégient une variable X par rapport aux autres et elles représentent les polynômes comme une somme de monômes en X dont les coefficients sont eux-mêmes des polynômes en représentation récursive.

On simule parfois cette représentation récursive. Pour un nombre n fixé à l'avance de variables $X_1, \ldots X_n$ on peut en effet obtenir une représentation de $\mathbf{A}[X_1, \dots X_n]$ sur un anneau $\mathbf{A}$ en travaillant d'abord dans $\mathbf{A}[X_1]$ qui est un anneau sur lequel on peut construire des polynômes en X_2, on obtient donc $(\mathbf{A}[X_1])[X_2]$ et ainsi de suite. On construit alors $(\dots(\mathbf{A}[X_1])\dots)[X_n]$. C'est la seule façon d'obtenir des représentations récursives en MAGMA.

En pratique, ces représentations ont l'inconvénient de ne pas être réellement creuses. Lorsque des calculs nécessitent beaucoup de variables, dont peu sont utilisées en même temps, cette technique peut s'avérer coûteuse. S'il y a n variables il y a toujours n niveaux de listes imbriqués. Par exemple pour $n = 3$ on représente 1 par

$$[(0, [(0, [(0, 1)])])]$$

où les crochets désignent les listes et les parenthèses les couples. Dans des programmes comme la clôture réelle de la première partie, n vaut souvent 1000, mais le niveau d'imbrication est limité à 4 ou 5.

AXIOM est un des rares systèmes permettant une véritable représentation récursive parce qu'on peut expliciter le type support lorsque l'on construit les polynômes récursifs. Les éléments sont dans un type récursif dont le cas de base est formé des éléments de l'anneau $\mathbf{A}$ et donc le cas inductif est formé d'un couple, dont le premier élément est le nom de la variable que l'on va utiliser et dont le deuxième élément est un polynôme en une variable à coefficient sur l'anneau des polynômes récursifs lui même.

Ainsi dans une syntaxe « à la FoC » :

```
species mult_polys(a_ring is ring)
rep =
  | Base in a_ring -> rep
  | Inductive in string -> univ_poly(self) -> rep
```
(6.1)

on donne en premier un nom de variable puis une définition de la valeur en tant que polynôme en une variable à coefficients sur les polynômes récursifs. Cette définition est bien sûr impossible en FoC, puisque la représentation doit être un type FoC et `self` n'est pas un type. Elle est par contre possible en AXIOM où la représentation est un autre domaine. On utilise le constructeur de domaine `Union` qui joue le rôle de constructeur de type somme. Les occurrences de `rep` à droite du signe « `=` » sont nommées « `%` » ainsi que `self`. On ne les explicite pas en AXIOM, donc la définition

```
Rep == Union(base : ARing,
             inductive : Tuple(String,UnivariatePolynomial(%)))
```

est « cohérente ».

On voit, dans la définition 6.1 des polynômes récursifs, que pour définir le support de l'espèce on utilise le nom `self` de manière abstraite. C'est la vision de `rep` comme ensemble mathématique offrant les opérations de polynômes. En FoC nous distinguons explicitement ces deux aspects. La définition des polynômes récursifs doit se faire en considérant la *représentation* des polynômes en une variable pour pouvoir construire les polynômes récursifs. Nous sommes ainsi amenés à considérer un constructeur de type `univ_poly_rep` afin de « déplier » l'occurrence de `self` qui désigne un type abstrait :

```
type univ_poly_rep('a) = alias list(int*'a)
```

ici, le type `univ_poly_rep('a)` est un synonyme pour `list(int*'a)`, ce qui permet de renommer un type existant. Nous pouvons maintenant écrire :

```
type mult_poly_rep('a) =
  | Base in 'a -> mult_poly_rep('a)
  | Inductive in string ->
    univ_poly_rep(mult_poly_rep('a)) ->
      mult_poly_rep('a)
```

Remarquons que nous avons pu donner la représentation des polynômes récursifs dans le système de types simples de FoC. Comme pour le prototype à base d'objets, il n'est pas nécessaire de disposer d'un système plus compliqué comme celui d'Axiom pour faire cela.

On peut ensuite donner explicitement l'espèce `polynomes_recursifs` qui va utiliser ce type :

```
species polynomes_recursifs(a is ring) inherits ring =

  rep = mult_poly_rep(a)
```

Nous voulons pouvoir interpréter un élément de la forme `Inductive`(V, P) de manière mathématique. C'est à dire en considérant que V est un nom de variable et que P est un polynôme en V, dont les coefficients sont en représentation récursive. Ceci nous permettra d'utiliser les opérations disponibles sur les polynômes en une variable pour effectivement écrire du code.

Nous sommes donc amenés à considérer une collection `up_rec` qui nous donne cette vision mathématique du type

```
univ_poly_rep(mult_poly_rep(rep_a))
```

où `rep_a` désigne la représentation des éléments de l'anneau `a_ring`. C'est le compilateur FoC qui gère le type `rep_a` implicitement à partir du paramètre de collection `a_ring`. Il nous faut définir la composante `up_rec` de notre espèce :

```
let up_rec is polynomes_univaries(self)
          = univ_poly(self)
```

Nous spécifions ici que nous utilisons les polynômes en une variable à coefficients sur `self` complètement implantés dans l'espèce `univ_poly`. Nous pouvons restreindre notre vision de `univ_poly(self)` aux opérations définies dans l'espèce `polynomes_univaries`. Évidemment, il faut que l'espèce complètement instanciée `univ_poly` utilise le constructeur de types `univ_poly_rep` pour que le typage soit cohérent, ce qui est vérifié par le compilateur de FoC.

On dispose ainsi d'une composante `up_rec` qui est une collection munie des opérations mathématiques des polynômes en une variable. Nous pouvons l'utiliser par l'appel `self !up_rec`, utiliser ses opérations (par exemple `self !up_rec !plus`) dans la définition d'autres composantes. Évidemment, ceci fait que les opérations de `self` qui sont utilisées par `univ_poly` doivent être définies sans utiliser `up_rec`. Comme ce n'est pas notre cas nous devrons préciser que toutes ces définitions sont récursives entre elles.

En effet, prenons l'exemple très simple d'affichage `print` d'un polynôme en plusieurs variables. Nous voulons utiliser l'opération `output`, d'affichage des polynômes en une variable, qui nous permet de nommer la variable. Cette opération utilise bien sûr l'opération `print` de l'anneau de ses coefficients soit `self` dans le cas présent. Ainsi `self !print` utilise `self !up_rec !output` qui utilise `self !print` créant une dépendance cyclique. Nous devons donc définir conjointement toutes les opérations qui utilisent `up_rec`. Encore une fois c'est le compilateur FoC qui vérifie ces dépendances.

On voit ainsi que les données que nous utilisons en calcul formel sont « simples » dans la mesure où on peut les décrire à l'aide d'un système de types comme celui de CAML-LIGHT ; on pourrait donc les écrire en C. C'est par contre leur interprétation en tant qu'ensemble mathématique qui est compliquée. Elle nous amène en effet à considérer récursivement une partie de la donnée comme un élément d'un ensemble mathématique avec toutes ses propriétés.

6.3 Généralisation

Dans ce qui précède nous n'avons jamais utilisé d'opérations spécifiques aux polynômes en une seule variable. Nous utilisons simplement les opérations de polynômes à coefficients sur un anneau commutatif, qui sont spécifiées dans l'espèce `polynomes_formels`. Il n'est pas nécessaire de se restreindre au cas d'une seule variable. La seule condition nécessaire est qu'on ait bien une arithmétique de polynômes dans laquelle on puisse prendre des degrés et les trier.

La figure 6.2 montre un exemple d'espèce ne définissant que les opérations `plus` et `print`. Elle utilise les définitions de la figure 6.1, qui implémente des sommes de termes indexées par un ensemble ordonné quelconque. On voit que, pour pouvoir faire des « simplifications » dans la représentation et enlever une variable « inutile », on doit utiliser la règle qui dit que λX^0 est mieux codé par λ. En général on implémente un anneau et pas un simple monoide additif (mais le code ne tient alors plus sur une page) et la simplification provient simplement de la règle $X^0 = 1$.

Conclusion

Si nous examinons le code des sommes de termes de la figure 6.2, pour pouvoir faire une simplification :

```
let r = !ind_self!plus(sum_x,sum_y) in
if deg_set!egal(ind_self!degre(r),deg_set!zero)
then ind_self!coefficient_dominant(r)
else #Inductive(s_x,r)
```

nous voyons que nous sommes obligés de tester si un élément est nul. Le calcul formel utilise souvent de tels tests d'égalité et on s'aperçoit rapidement que l'on doit la spécifier très tôt dans la hiérarchie. La librairie FoC reprend en cela la librairie Axiom.

On peut toutefois parfaitement parler de l'égalité sans en avoir une implémentation constructive. Par exemple, en OCaml l'égalité des fermetures n'est pas définie. Elle a toutefois un sens précis puisque le compilateur remplace souvent un appel $F(x)$ par un appel $F_o(x)$ où F_o est une version optimisée de la fermeture F qui calcule le même résultat. Le modèle FoC ne fait pas d'hypothèses sur l'égalité qui est laissée sous la responsabilité du programmeur. C'est donc bien la librairie de calcul formel qui affirme que cette égalité doit être décidable. Je voulais pouvoir me placer dans ce cadre habituel pour mieux considérer les problèmes. Il pourrait bien sûr être intéressant de voir dans quelle mesure cette contrainte peut être levée.

Dans cette partie, nous avons vu que l'effort du projet FoC s'est concentré sur les assises sémantiques d'un langage dédié pour le calcul formel. Comme nous voulions mettre l'accent sur la certification, le code FoC est analysé par le compilateur qui produit des « dépendances » entre les méthodes d'une espèce. La figure 6.3 montre l'analyse des dépendances entre les méthodes de l'espèce `ordre_monomial` (voir [Pre01, PD03]). Le compilateur est capable de

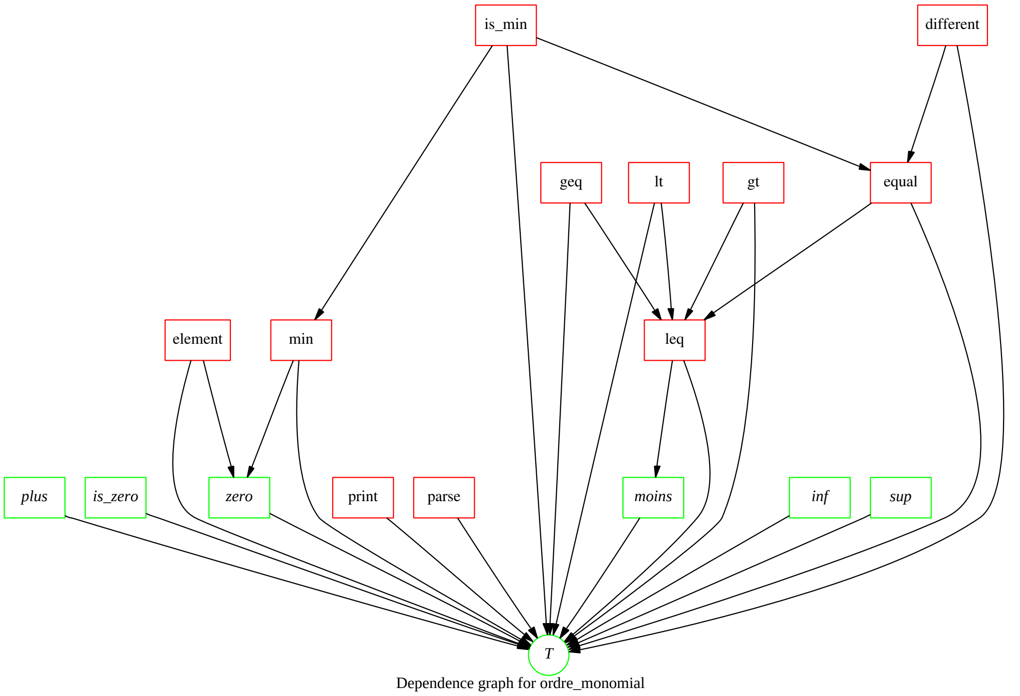
FIG. 6.1 – Somme de termes en FoC

```
type liste_indexee('a,'d) = alias list('a*'d)

species somme_indexee(a is monoide_additif, d is ordre_monomial)
  inherits monoide_additif =

  rep = liste_indexee(a_mon,deg_set);

  let zero = #Nil ;
  let lift(a) =
    if a_mon!egal(a,a_mon!zero)
    then !zero
    else #Cons(a,deg_set!zero);
  let degre(x) = ... ;
  let coefficient_dominant(x) = ... ;
  let egal(x,y) = ... ;
  let output(s,v) = ... ;

  let rec plus(x,y) =
   match x with
   | #Nil -> y
   | #Cons(h_x,t_x) -> (* h_x is the monomial, t_x is the rest *)
     match y with
     | #Nil -> x
     | #Cons(h_y,t_y) -> (* h is head and t is tail *)
       let c_x = #fst(h_x) (* the coefficient *) in
       let d_x = #fst(h_x) (* the degree *) in
       let c_y = #fst(h_y) (* the coefficient *) in
       let d_y = #fst(h_y) (* the degree *) in
       if deg_set!lt(d_x,d_y) (* d_x < d_y *)
       then #Cons(h_y,!plus(x,t_y))
       else
         if deg_set!lt(d_y,d_x) (* d_y < d_x *)
         then #Cons(h_x,!plus(t_x,y))
         else (* d_x = d_y *)
           let c = a_mon!plus(c_x,c_y) in
           if a_mon!egal(c,a_mon!zero)
           then !plus(t_x,t_y)
           else #Cons(#crp(c,d_x),!plus(t_x,t_y))
     end
   end ;
```

FIG. 6.2 – Sommes récursives en FoC

```
type indexed_by_strings('a,'d) =
  | Base in 'a -> indexed_by_strings('a)
  | Inductive in string ->
     liste_indexee(indexed_by_strings('a,'d), 'd) ->
      indexed_by_strings('a);;
species somme_de_termes_recursive
     (a_mon is monoide_additif,
      deg_set is ensemble_ordonne)
  inherits monoide_additif =
  rep = indexed_by_strings(a_mon);
  let zero = #Base(a_mon!zero);
  let rec
      ind_self = somme_indexee(elf, deg_set)
  and print(x) = match x with
      | #Base a -> a_mon!output(a)
      | #Inductive(name, value) ->
        !ind_self!output(value,name)
      end
  and plus(x,y) = match x with
     | #Base(a) -> match(y) ->
       | #Base(b) -> #Base(a+b)
       | #Inductive(s_y, sum_y) ->
         #Inductive(s_y,ind_self!plus(ind_self!lift(x),sum_y))
       end
     | #Inductive(s_x, sum_x) -> match(y) ->
       | #Base(b) ->
         #Inductive(s_y,ind_self!plus(sum_x,ind_self!lift(y)))
       | #Inductive(s_y, sum_y) ->
         if #str_lt(s_x,s_y)
         then #Inductive(s_y,ind_self!plus(ind_self!lift(x),sum_y))
         else
           if #str_lt(s_y,s_x)
           then #Inductive(s_x,ind_self!plus(sum_x,ind_self!lift(y)))
           else
            let r = !ind_self!plus(sum_x,sum_y) in
            if deg_set!egal(ind_self!degre(r),deg_set!zero)
            then ind_self!coefficient_dominant(r)
            else #Inductive(s_x,r)
     end;
```

FIG. 6.3 – Les ordres monomiaux

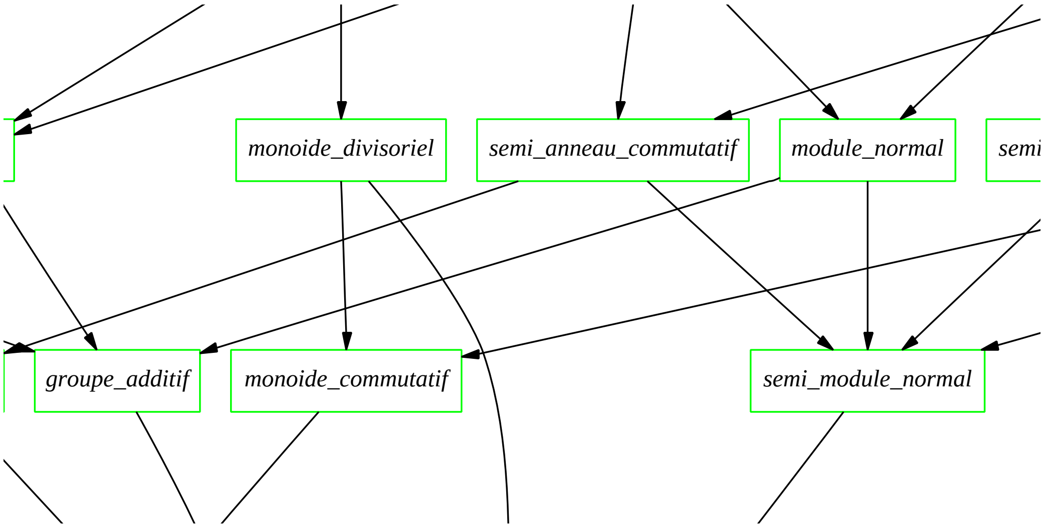


vérifier les dépendances entre les espèces de la librairie. La figure 6.4 montre une petite partie du graphe d'héritage. Le compilateur vérifie bien sûr l'absence de cycles dans ce graphe. Le projet a ainsi obtenu des outils qualitatifs pour aider le développeur de calcul formel a concevoir ses programmes. Ces outils doivent maintenant avoir une véritable interface utilisateur qui puisse aider le développeur dans son travail.

La syntaxe concrète de FOC est bien sûr moins riche que celle d'AXIOM. Elle est toutefois suffisamment expressive pour qu'un stagiaire de DEA ait pu traduire d'AXIOM en FOC l'ensemble du modèle de la clôture réelle d'AXIOM (voir la section 2). Un autre stagiaire a pu implanter l'algorithme de Cantor Zassenhauss (voir [QN98]) de factorisation de polynômes en une variable dans les corps finis. Ces stagiaires étaient des informaticiens formés à OCAML et appréhender FOC leur a été plus facile qu'à un mathématicien qui ne connaît pas toujours la programmation fonctionnelle.

De mon point du vue de programmeur de calcul formel on peut déjà avoir « confiance » dans un code écrit en FOC. Même en l'absence de certification des algorithmes, des vérifications importantes sont faites par le compilateur FOC. L'analyse du code FOC est basée sur un modèle sémantique certifié en COQ et le code OCAML produit est encore vérifié par le compilateur OCAML.

FIG. 6.4 – Dépendances de la librairie



Un éventuel bug du compilateur FoC dans la cohérence du code peut être détecté par le compilateur OCAML qui sert ainsi de « garde fou ».

Conclusion générale

Dans ce document, j'ai d'abord présenté le cadre de mes résultats de calcul formel. Les plus importants sont :

- Le modèle abstrait de la section 2.2 qui permet de faire des calculs dans des tours d'extensions. On peut ainsi calculer dans des extensions algébriques réelles indépendamment des algorithmes que l'on utilise pour « déterminer » les signes.
- L'algorithme 4 et la proposition 4 de la section 3.2 sont le moteur des méthodes de résolution d'équations à base « d'ensembles triangulaires » de Daniel Lazard. Aujourd'hui les programmes ALDOR de Marc Moreno commencent à offrir une alternative crédible aux « bases de Gröbner ».
- La généralisation du théorème classique de Sturm est la proposition 5. L'algorithme 5 permet en pratique d'éviter de scinder les intervalles d'isolation. Nous généralisons ainsi les méthodes de Marie Françoise Roy qui ont une bonne complexité théorique.

  De même que l'algorithme 4 a eu des applications pour la résolution d'équations, on peut penser que l'algorithme 5 puisse s'appliquer à la résolution réelle d'équations. On peut imaginer une alternative à la « représentation univariée rationnelle » de Marie Françoise Roy et Fabrice Rouillier.

Il faut encore faire de ces algorithmes et du prototype qui les valide un véritable logiciel qui puisse être distribué. Récemment, sur l'initiative de Tim Daly l'un des implémenteurs d'AXIOM, la société NAG a abandonné ses droits sur AXIOM. Il est en passe d'être diffusé sous licence libre. Ceci permettra à la communauté de calcul formel de capitaliser un effort représentant 300 hommes années. Une nouvelle version de la clôture réelle pour AXIOM est envisageable.

Les représentations à base d'intervalles d'isolation pour les nombres algébriques réels de la section 2.2.3 sont bien adaptées aux nombres réels usuels. Dans certains cas, la géométrie réelle nécessite de se placer dans un voisinage d'un point. On fait une « petite déformation » d'une courbe algébrique, on ajoute ainsi un « ϵ » infiniment petit au corps de base. Pouvoir considérer que ce corps contenant des nombres infiniment grands et infiniment petits est réel clos apporte une aide au mathématicien. Il peut ainsi prendre les racines de polynômes dont les coefficients contiennent des infinitésimaux.

Actuellement, aucun programme ne permet de gérer efficacement de tels nombres. Ce sont en effet des séries de Puiseux très lourdes à manipuler. Une généralisation du modèle de la clôture réelle permet d'en avoir une représentation plus compacte puisque seules les « racines réelles » sont des séries. Les nombres algébriques eux mêmes sont des polynômes comme dans le chapitre 2. J'ai réalisé une première implémentation en AXIOM de cette arithmétique qui montre qu'on peut avoir une bonne efficacité.

Il faut encore formaliser la théorie, mais les infinitésimaux sont un premier pas vers une « analyse non standard » effective. Les techniques de simplifications du théorème 5 peuvent s'appliquer plus largement que dans les anneaux contenant des nombres algébriques. Certaines s'appliquent aux infinitésimaux.

Une question importante en calcul formel est la résolution des équations en dimension positive. Les solutions du système ne sont alors plus des points mais bien des courbes ou des surfaces. On pourrait peut être appliquer certaines techniques de simplifications des infinitésimaux pour les « variables transcendantes » qui interviennent au cours des calculs.

L'implémentation de ces « calculs » compliqués m'a naturellement amené à la seconde partie de ce document. J'ai été amené à me poser en termes précis des questions sur la « sémantique » des programmes que j'écrivais. C'est ce qui a aboutit à la création, avec Thérèse Hardin, du projet FoC et aux concepts sous-jacents de la section 5 du langage FoC. La section 6 a montré que l'on pouvait déjà écrire des programmes dans lesquels on peut avoir « confiance ».

Je pense qu'il est donc déjà possible de programmer dans un tel langage. En particulier, on peut maintenant donner un véritable cadre aux « mathématiques effectives » que nous faisons en calcul formel. J'espère qu'une prochaine diffusion de FoC pourra montrer que l'on peut concevoir et démontrer des algorithmes dans un même formalisme. Nous espérons ainsi recueillir des éléments permettant de mieux adapter la syntaxe concrète de notre langage aux habitudes des mathématiciens.

Avec FoC on a une vision uniforme de la « construction » mathématique. Elle va de la description d'une structure algébrique ou d'un algorithme à son implantation, « documentée » par ses propriétés. La différence entre calculer et prouver s'estompe. La preuve du théorème est « ce que fait » l'algorithme et prouver l'algorithme « est » le théorème.

Bien souvent, en calcul formel, nous sommes amenés à concevoir un cadre théorique pour montrer que les calculs que nous faisons on bien un sens.

D'une certaine façon, nous construisons un nouvel objet mathématique et notre problème est de lui donner un « sens ». Nous montrons ainsi qu'il est fondé. De telles approches nécessitent un environnement comme celui de FoC pour que le mathématicien puisse parler de la même façon de la construction et de sa justification.

Actuellement, la preuve est réservée à la phase de développement et de compilation. On pourrait envisager de faire une preuve en cours d'exécution, pour tirer dynamiquement partie d'une propriété par exemple. On peut aussi imaginer de faire un calcul pendant une preuve. Les algorithmes « d'élimination de quantificateurs » en sont un exemple, la « décomposition cylindrique » de Georges Collins permet d'aborder ces problèmes.

Nous allons bientôt pouvoir aborder des exemples importants comme l'« évaluation dynamique » de Dominique Duval, dont nous avons « esquissé » le fonctionnement dans le chapitre 2. Nous avons là un procédé constructif dont le sens peut être éclairé par la théorie des esquisses. Les « anneaux de Prüfer », considérés par Henri Lombardi, offrent un cadre mathématique où la résolution d'équations est naturelle. Une vision constructive de ces anneaux pourrait permettre des progrès dans la résolution d'équations. Ces approches « constructives » des mathématiques pourront ainsi être automatisées.

Au delà du calcul formel, FoC offre au programmeur une approche homogène pour la représentation des données, les opérations que l'on peut leur appliquer et leurs nécessaires spécifications. La programmation par objets permet au programmeur d'abstraire la représentation pour n'exposer que des méthodes. Les variables d'instance cachent cette représentation. La programmation « par représentation », que nous proposons avec FoC, permet elle de considérer la donnée effective comme un paramètre implicite. On peut ainsi énoncer plus facilement les spécifications puisqu'elles ont alors le même statut que les opérations.

Le paradigme des entités, collections et espèces que nous proposons est directement issu des mathématiques. Cette vision est donc très naturelle et on peut penser que le modèle s'applique plus largement. Ainsi, un physicien théorique considère des « particules » qu'il manipule comme des « formules ». Beaucoup d'entre eux utilisent déjà des outils de calcul formel pour faire ces « calculs ». On peut penser qu'une meilleure description du formalisme spécifique qu'ils utilisent leur permettrait de faire plus de calculs automatiquement.

Beaucoup de livres scientifiques sont écrits à la manière des mathématiques. On y développe d'abord une théorie formelle que l'on applique ensuite. FoC est adapté à cette démarche et pas simplement au calcul formel. On peut penser que la demande croissante de fiabilité des logiciels amènera rapidement un changement dans leurs méthodes de développement. Il faudra, en particulier, mettre plus l'accent sur la spécification et sa cohérence. Pouvoir manipuler la spécification et l'implantation dans le même formalisme apporte une aide au développement. On écrit ainsi plus rapidement des programmes justes.

La méthodologie de conception de FoC a été celle d'une écoute permanente. Le spécialiste du calcul formel que je suis à été obligé de prendre en compte les nécessités de la sémantique et de la certification de programmes. De même, il a fallu intégrer la spécification mathématique dans la sémantique du langage. Les experts en certification ont ainsi pu prendre en compte les nécessités de l'algébriste qui élabore une théorie par étapes.

Le modèle uniforme que nous proposons dans FoC permet d'aborder l'algèbre des mathématiciens de la même manière que la logique des sémanticiens. Je pense qu'un tel langage peut trouver des applications au delà des domaines du calcul formel et de la certification dont il est issu.

Bibliographie

[AC96] M. Abadi , L. Cardelli. *A Theory of objects*. Springer, 1996.

[Ale98] G. Alexandre. « *d'Axiom à Zermelo* ». Thèse de doctorat, Laboratoire d'Informatique de Paris 6, 1998.

[ALM99] P. Aubry, D. Lazard, , M. Moreno Maza. « On the theories of triangular sets ». *Journal of Symbolic Computation, Special Issue on Polynomial Elimination*, 28 :105–124, 1999.

[BBD$^+$] M. Bronstein, W. Burge, T. Daly, P. Gianni, J. Grabmeier, , R. Jenks. « *AXIOM User Guide* ». NAG.

[BCL83] B. Buchberger, G.E. Collins, , R. Loos, Éditeurs. *Computer Algebra Symbolic and Algebraic Computations*. Springer Verlag, 1983.

[BCR87] J. Bochnak, M. Coste, , M.F. Roy. *Géométrie algébrique réelle*. Springer-Verlag, 1987.

[Beu98] F. Beukers. Factorization of Polynomials. dans Cohen et al. [CCS98], pages 78–90.

[BHR99] S. Boulmé, T. Hardin, , R. Rioboo. « Polymorphic data types, objects, modules and functors : is it too much ? ». Rapport Technique, Laboratoire d'Informatique de Paris 6, 1999. `http://www.lip6.fr/reports/lip6.1999.012.html`.

[Bou00] S. Boulmé. « *Spécification d'un environnement dédié à la programmaion certifiée de bibliothèques de Calcul Formel* ». Thèse de doctorat, Université Paris 6, 2000.

[BPR96] S. Basu, R. Pollack, , M. F. Roy. « On the Combinatorial and Algebraic Complexity of Quantifier Elimination ». *Journal of the ACM*, 43(6) :1002–1046, 1996.

[BT71] W. S. Brown , J. F. Traub. « On Euclid's Algorithm and the Theory of Subresultants ». *Journal of the ACM*, 1971.

[CCS98] A. Cohen, H. Cuyper, , H. Sterk, Éditeurs. *Some Taps of Computer Algebra*, volume 4 de *Algorithms and Computation in Mathematics*. Springer, 1998.

[CM98] G. Cousineau , M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[Col67] G. E. Collins. « Subresultants and Reduced Polynomial Remainder Sequences ». *Journal of the ACM*, pages 128–142, 1967.

[Col75a] G.E. Collins. « Quantifier Elimination For Real Closed Fields By Cylindrical Algebraic Decomposition ». *Lecture Notes In Computer Science*, 33 :134–183, 1975.

[Col75b] G.E. Collins. « Quantifier Elimination For Real Closed Fields By Cylindrical Algebraic Decomposition ». *Lecture Notes in Computer Science*, 33 :134–183, 1975.

[CR88] M. Coste , M.F. Roy. « Thom's Lemma, the coding of Real Algebraic Numbers and the Computation of the Topology of Semi-Algebraic Sets ». *Journal Of symbolic Computation*, 5 :121–129, 1988.

[Cro99] Projet Croap. « calcul Formel et Déduction ». URL, 1999. `http://www.inria.fr/rapportsactivite/RA98/croap/resul_cfc.html`.

[DDDD85] J. Della Dora, D. Discrescenzo, , D. Duval. « About a new method method for computing in algebraic number fields ». *Lecture Notes in Computer Science*, 204, 1985.

[DGJ+84] J. Davenport, P. Gianni, R. Jenks, V. Miller, S. Morrison, M. Rothstein, C. Sundaresan, R. Sutor, , B. Trager. « *Scratchpad* ». Mathematical Sciences Department, IBM Thomas Watson Research Center, 1984.

[DGV96] D. Duval , L. Gonzalez Vega. « Dynamic Evaluation and Real Closure ». *Mathematics and Computers in Simulation*, 42 :551–560, 1996.

[DST87] J. Davenport, Y. Siret, , E. Tournier. *Calcul Formel : systèmes et algorithmes de manipulations algébriques*. Masson, 1987.

[Fechter01] Stéphane Fechter. « Une sémantique pour FoC ». Rapport Technique, Laboratoire d'Informatique de Paris 6, 2001. `http://www-spi.lip6.fr/~fechter/`.

[Fil99] J.-C. Filliâtre. « *Preuve de programmes impératifs en théorie des types* ». Thèse de doctorat, Université Paris-Sud, July 1999.

[GCL93] K. O. Geddes, S. R. Czapor, , G. Labahn. *Algorithms for Computer Algebra*. Kluver Academic Publishers, 1993.

[GVRR98] L. Gonzalez Vega, M-F. Roy, , F. Rouillier. Symbolic Recipes for Polynomial System Soloving. dans Cohen et al. [CCS98], pages 34–65.

[GVRRT98] L. Gonzalez Vega, F. Rouillier, M-F. Roy, , G. Trujillo. Symbolic Recipes for Real Solutions. dans Cohen et al. [CCS98], pages 121–167.

[Hol41] A. Hollkott. « *Finite Konstruktion geordneter algebraischer Erweterungen von Geordneten Grundkorpern* ». Dissertation, Univ Hamburg, 1941.

[Lan64] S. Lang. *Algebraic Numbers*. Addison-Wesley Pub. Co., 1964.

[Lan69] S. Lang. *Algebra*. Addison-Wesley Pub. Co., 1969.

[Lan90] L. Langmyr. « Algorithms for a multiple algebraic extension ». dans *Effective methods in algebraic geometry 90*, volume 94 de *Progress in mathematics*, pages 235–248. Birkhauser, 1990.

[LDG+00] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, , J. Vouillon. « *The Objective Caml system release 3.00 Documentation and user's manual* ». INRIA, 2000. `http://pauillac.inria.fr/ocaml/htmlman/`.

[Lom02] H. Lombardi. « Dimension de Krull, Nullstellensatze et Évaluation Dynamique ». *Math. Zeitschrift*, 2002.

[Loo83a] R. Loos. Computing in Algebraic Extensions. dans Buchberger et al. [BCL83], pages 173–188.

[Loo83b] R. Loos. Generalized Remainder Seuqnences. dans Buchberger et al. [BCL83], pages 115–138.

[Loo83c] R. Loos. Real Zeros of Polynomials. dans Buchberger et al. [BCL83], pages 83–94.

[LR01] T. Lickteig , M-F. Roy. « Sylvester-Habicht sequences and fast Cauchy index computations ». *Journal Of symbolic Computation*, 31 :315–341, 2001.

[LRR96] Z. Ligatsikas, R. Rioboo, , M. F. Roy. « Generic Closure Of an Ordered Field, Implementation in Axiom ». *Matematics and Computers in Simulation*, 42 :541–549, 1996.

[LRSED00] H. Lombardi, M.F. Roy, , M. Safey El Din. « New Structure Theorems for subresultants ». *Journal Of symbolic Computation*, 29 :663–690, 2000.

[MM97] M. Moreno Maza. « *Calculs de Pgcd au dessus des Tours d'Extensions Simples et Résolution des Systèmes Algébriques* ». Thèse de doctorat, Laboratoire d'Informatique de Paris 6, juin 1997.

[MMR96] M. Moreno Maza , R. Rioboo. « Polynomial Gcd Computations over Towers of Algebraic Extensions ». *Lecture Notes in Computer Science*, 948, 1996.

[PD03] V. Prevosto , D. Doligez. « Inheritance of Algorithms and Proofs in the Computer Algebra Library Foc ». *Journal of Automated Reasoning*, 2003. Sous presse, Special Issue on Mechanising and Automating Mathematics.

[PDH02] V. Prevosto, D. Doligez, , T. Hardin. « Algebraic Structure and Dependent Records ». dans Sofiène Tahar César Muñoz , Víctor Carreño, Éditeurs, *Proceedings of TPHOLs 02*. Springer-Verlag, August 2002.

[Pot] Loic Pottier. « Action Calcul Formel Certifié ». URL. `http://www-sop.inria.fr/croap/CFC/ programme.html`.

[Pre00] V. Prevosto. « Vers une interface utilisateur pour Foc ». Rapport de DEA, Université Paris 6, Septembre 2000.

[Pre01] V. Prevosto. « Prototype d'interface utilisateur de la librairie Foc ». dans *Journées Francophones de Langages Applicatifs*, janvier 2001.

[PT00] E Poll , S. Thompson. « Integrating Computer Algebra and Reasoning through the Type System of Aldor ». dans Hélène Kirchner , Christophe Ringeissen, Éditeurs, *Frontiers of Combining Systems : Frocos 2000*, volume 1794 de *Lecture Notes in Computer Science*, pages 136–150. Springer, 2000.

[QN98] C. Quitté , P. Naudin. « Univariate polynomial factorization over finite fields ». *Theoretical Computer Science*, 191(1–2) :1–36, 1998. Tutorial.

[Rém02] D. Rémy. Using, Understanding, and Unraveling the OCaml Language. dans G. Barthe, Éditeur, *Applied Semantics. Advanced Lectures. LNCS 2395.*, pages 413–537. Springer Verlag, 2002.

[Rio] R. Rioboo. « The RECLOS distribution ». Available on the Web. `http://ftp.lip6.fr/lip6/softs/formel/Axiom/RealClosures/`.

[Rio91] R. Rioboo. « *Quelques aspects du calcul exact evec les nombres réels* ». Thèse de doctorat, Laboratoire d'Informatique Théorique et Programmation, 1991.

[Rio02] R. Rioboo. « Towards faster Real Algebraic Numbers ». dans *International Symposium on Symbolic and Algebraic Computations*, 2002.

[Sal02] M. Salou. « *Théorie algorithmique des anneaux arithmétiques, des anneaux de Prüfer et des anenaux de Dedekind* ». Thèse de doctorat, Université de Besançon, 2002.

[Str97] A. Strzeboński. « Computing in the Field of Complex Algebraic Numbers ». *Journal Of symbolic Computation*, 24 :647–656, 1997.

[Tho] S. Thompson. « Atypical Group ». URL. `http://www.cs.ukc.ac.uk/people/ staff/sjt/`.

[Vou98] J. Vouillon. « Using modules as classes ». dans *Informal proceedings of the FOOL'5 workshop*, 1998. `http://pauillac.inria.fr/~remy/fool`

[Wad90] P. Wadler. Comprehending Monads. dans *LISP'90, Nice, France*, pages 61–78. ACM Press, 1990.

[Zwa99] J. Zwanenburg. « *An Object Oriented Programming Logic Based on Type Theory* ». Dissertation, Eindhoven University of Technology, 1999.